Oblig5

Introduction:

The goal of the fourth assignment was to create a parallel algorithm for creating a convex hull of a set of points. There are multiple ways to do this, but this solution used the “Divide and Conquer” technique. The set of points is divided into smaller subsets, where each subset gets its own convex hull. When all threads are finished creating their hulls, their points are the basis set for the complete convex hull, found sequentially. The assignment includes both a sequential and parallel implementation, and their runtimes are compared.

Usage:

The program is compiled with the terminal command:

*javac *.java*

The program is run with the terminal-command:

java Oblig5 <n> <seed> <mode>

where <*n*> is the size of the matrix, <seed> is the random number generator to use and <*mode*> is the implementation you want run. The modes are <*seq*> and <*par*>.

The number of threads on the computer will be used.

Example:

java Oblig5 1000 4 par

Will run the parallel implementation on n = 1000 (7 times) with the seed 4, using 8 threads.

Note:

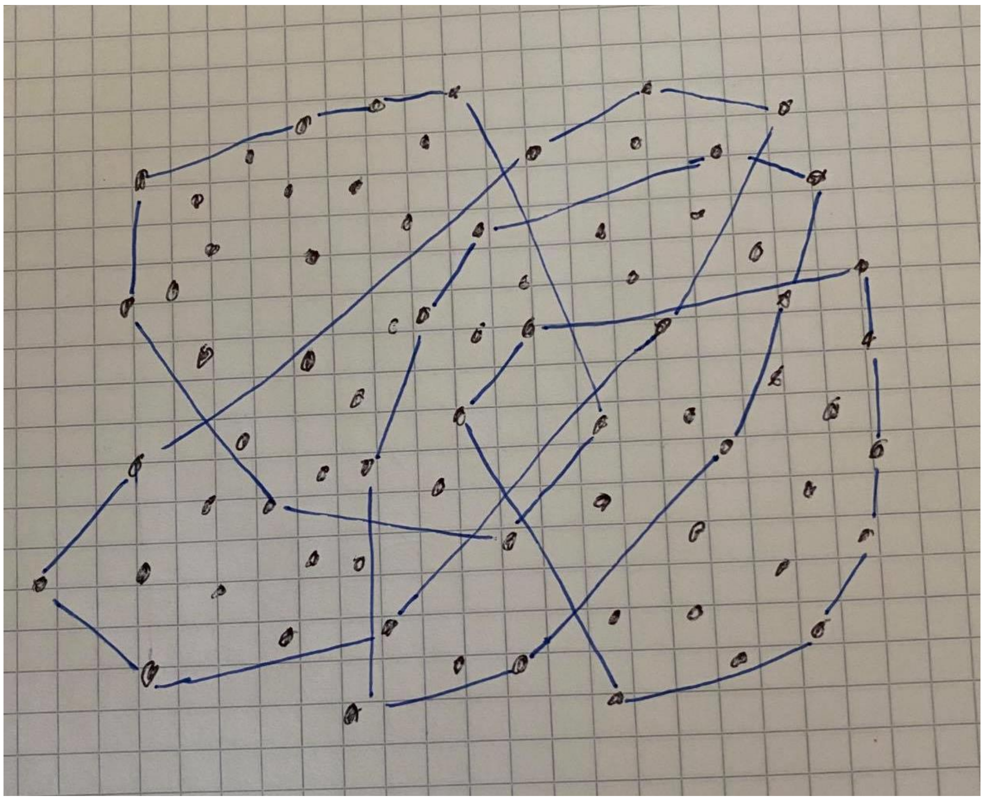
Both implementations present the convex hull graphically if n <= 1000.

For all the different numbers-n tested <= 100 000, results are stored in text files. They are named *CONVEX-HULL-POINTS_<n>.txt*

Parallel Version:

For the parallel implementation option 1 was chosen, divide and conquer. The set of points are evenly divided between the threads. Each thread finds their own min_x and max_x and build their own convex hull. The points on each hull are added to a global IntList in a synchronized method, together with each thread's min_x and max_x values. If a local min/max is smaller/greater than the global, the value is updated. The threads are synchronized with a cyclic barrier.

Example where 4 threads each have been given ¼ of the points in the set, and have build their own convex hulls:



When all the threads are finished the main method will create the final convex hull than encapsulates all the points in the set. This is done very efficiently since it only works on the few points that make up all the smaller (local) convex hulls, and the min_x and max_x values are already found.

The parallel solution uses the same findPointsToLeft-method as the sequential solution, with no modifications needed. When there are multiple points on a line that should be included in the hull they are added to their own list, sorted, then added to the hull.

Measurements:

Processor: Intel Core i7-8550U CPU @ 1.80GHz 1.99 GHz, 8 cores

RAM: 24,0 GB (23,9 GB usable)

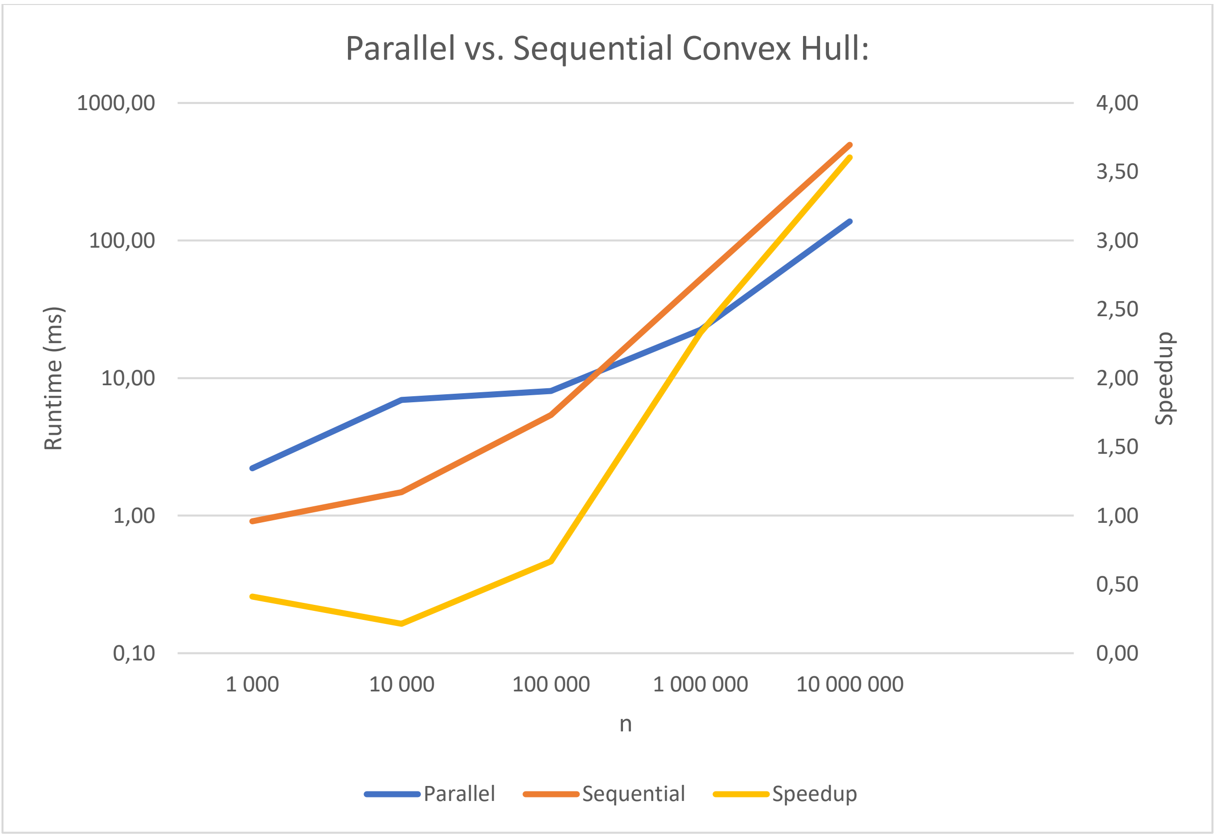
64-bit operating system, x64-based processor.

All runtimes are the median of 7 calls, using 8 threads.

Convex Hull – seed=4, threads=8:

n	Parallel (ms)	Sequential (ms)	Speedup
100	1,70	0,12	0,07
1 000	2,21	0,91	0,41
10 000	6,92	1,48	0,21
100 000	8,08	5,39	0,67
1 000 000	22,41	52,20	2,33
10 000 000	137,72	496,37	3,60

Convex Hull- The Runtimes and Speedup for each n:

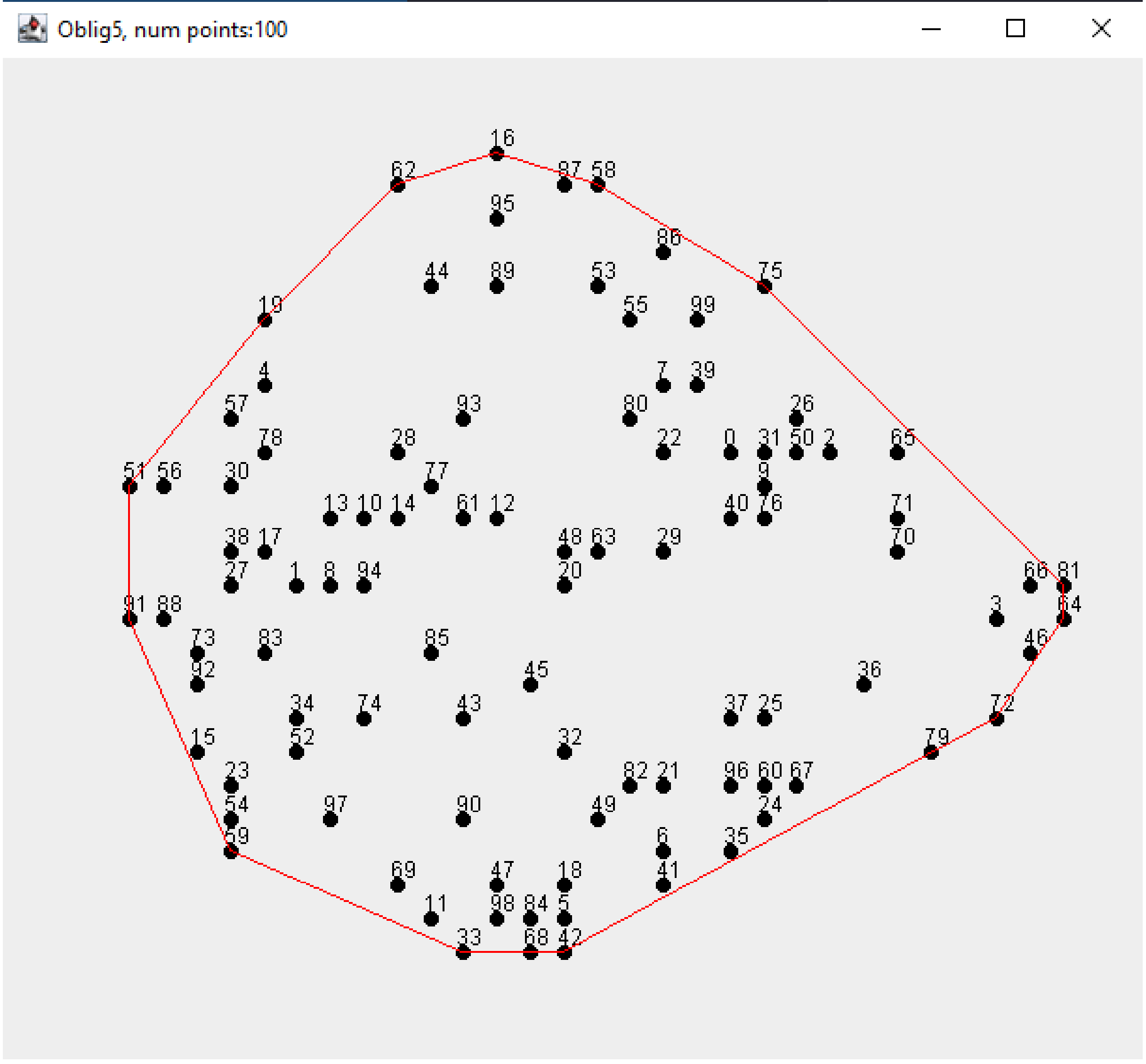


It takes a large set for any significant speedup. Around 100 000 points the parallel solution can sometimes be faster. For 1 million points and up it is significantly faster to do this in parallel.

The parallel implementation was also tested with multiple threads per core, but it was not any faster than keeping it one thread per core.

Each thread works on a much smaller set of points, requiring less comparisons and recursions. It does not matter than the different threads have convex hulls crossing eachother. When all threads are done making their local hull and adding it to the global IntList, the final sequential step (making the complete convex hull) only has to work on a very small number of points. For small *n* a big portion of all the points will be included in the local hulls and not a lot of work is spared. On the other hand, for large *n* a lot of recursive work is spared.

The convex hull for n=100, seed=4:



Conclusion:

Implementing a parallel convex hull algorithm is worth it for large sets of points, and does not require a lot of modifications or additions to the sequential code.