Report

Eirik Eggset

Introduction

In this assignment I have made a java program to find the convex hull of a set of points in a 2d space. I have implemented both a sequential and parallel version. The main program runs both the sequential and parallel implementations and compares the runtimes. It also has a testprogram for the speedup. The variable for the number of runs can be changed, it is at the top of the file Oblig4.java.

User guide

To run the program, you use the command: java Oblig4 *[flags] <n> [seed]

- The flags are optional, to include them separate them with spaces between before <n>. A list of the flags:
    - **-w**: writes the results to a file using the precode.
    - **-d**: draws a graphical representation of the results. One for every run. (not recommended when running test program). Uses the precode.
- **n**: the number of points. If **n** is set to -1 the test program is run, this is explained in the measurements section.
- **seed**: determines the seed used for generating the points.

Implementation

To make this program I made a class 'ConvexHull' implementing the functions as recommended in the oblig task. This includes: 'findHullSeq()' and 'seqReq()'. Originally I made a list and added all values to that list recursively, however this would be less efficient to parallelize. Therefore I changed the sequential algorithm so that it is as similar to the parallel version. In this implementation every recursive call instead returns the result of its recursive calls, where the base case is the points on the line (apart from corners, if there are any). This costs a bit more memory, but is good enough.

I follow the algorithm as described in the assignment. The only difference is when finding all the values along the convex hull, and not only the corners. This is because the assignment had a very unclear and confusing description of this part of the algorithm. As I know many others struggled with this specific part, I recommend that you reformulate this part of the assignment till next year. The implementation I used was simple, but might not be as efficient as using recursion, which was what it seemed like you wanted. When finding all the points with the largest negative distances I made a list with all the points where distance=0 in the function 'findLargestNegativeDistance()'. This function returns the negative point the furthest away. If no point is further away than 0 it means you have reached the top of the recursive tree, in which case it returns -1. When this happens 'seqRec()' no longer makes a recursive call, instead it sorts the points with distance 0 by distance from p2 (p2 as in the right point of either the left or right group), and then adds the points.

In order to sort the IntList I made a function sort(int relativePoint) which uses a modified version of quick sort to sort the points by relative distance to 'relativePoint', in descending order.

Parallel implementation

The algorithm

In order to parallelize the algorithm, I used the Libraries ‘ExecutionerService’ to start and run the threads, and made worker classes implementing the ‘Callable’ interface. I chose to parallelize in the function ‘ParRec()’, which is the recursive function. However I choose to only make parallel calls in the first calls on each side of the algorithm (with side I mean when I use two separate recursive calls to calculate the top and bottom part of the hull).

The first part of the algorithm is a call to ‘findHullPar()’. This takes an optional parameter, which is the number of parallel recursion layers. (If no parameter is used the default value is used, which is the number of cores in your processor. In my program the default is always used, but the option is there). In this function I make a recursion depth which is the number of recursion layers divided by two, and rounded down in order to make an even number. The reason I divide by two is to give half the layers as parameter ‘recursionDepth’ to the top call and half to the bottom call.

In the parRec I make an object of the class ParRecWorker, which implements the Callable interface. I return this object.

The ParRecWorker is almost identical to the sequential recursive function. The difference is if recursion depth is 0 it starts making sequential calls to build the coHull instead. In the constructor the ParRecWorker decreases its recursion depth by 2. As the Callable function ‘call()’ returns a Future object the function waits for the object (and therefore its recursive calls) to finish before building and returning the list. If recursion depth is 0 the sequential recursive function is called, therefore it can just add the result directly to the list.

Future class, Callable interface and ExecutionerService

I used classes from the java concurrent library.

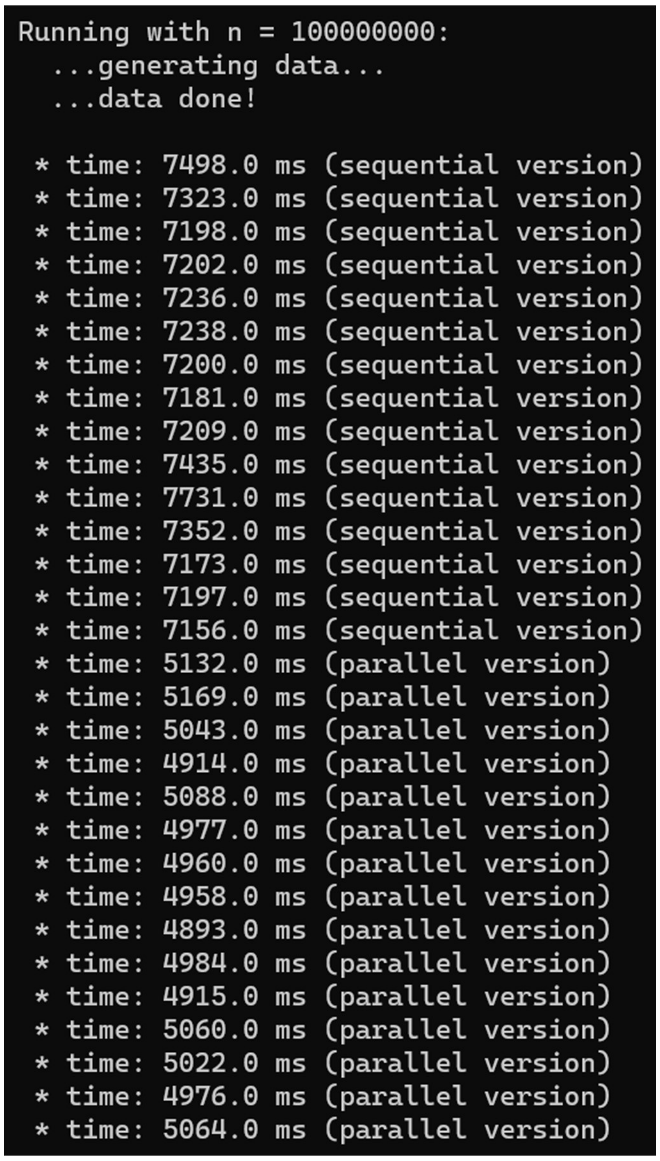
The worker class implements from the Callable interface. This class is designed for parallel solutions and implements the function ‘call()’ returns a value of the respective type, wrapped in a Future object. When setting a variable as a future object the program does not wait for ‘call()’ to finish. Instead it continues. When you want to use the IntList wrapped inside the Future object there is a function ‘get()’. This makes the program wait until the respective ‘call()’ is made.

Each parReq() where recursionDepth > 0 calls get() at the end of the function. But since we are using the ExecutionerService the thread is not blocked. This is because the ExecutionerService creates a pool of parallel tasks which are run in a round robin fashion. Therefore waiting tasks will just be skipped while waiting.

Measurements

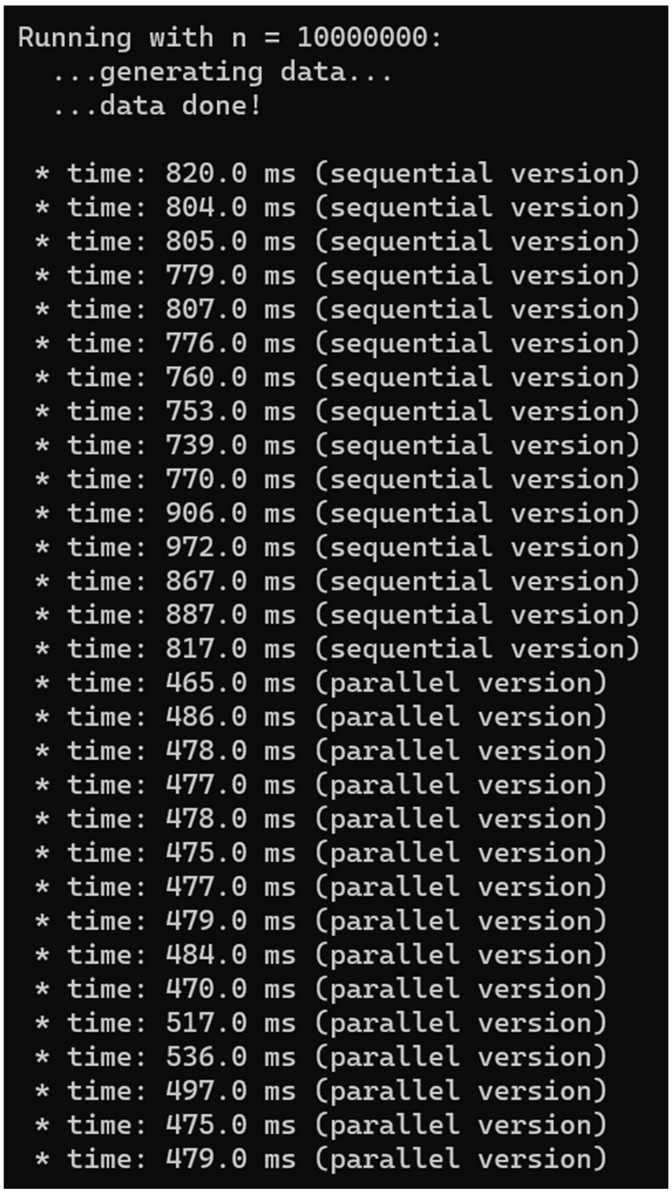
If the program is run with n=-1 the testprogram will run. It runs the algorithms for n values: [1000, 10000, 100 000, 1 000 000, 10 000 000, 100 000 000], 16 times each. All runtimes are printed as they finish. In the end the mean speedup is printed for each value of n.

The speed requirements was that the algorithm runs below 2s on 10 000 000 points, my worst runtime of the 15 runs was 0.97s. The requirement for 100 000 000 was 14s, my worst runtime was 7.73s. This was for the sequential runs.



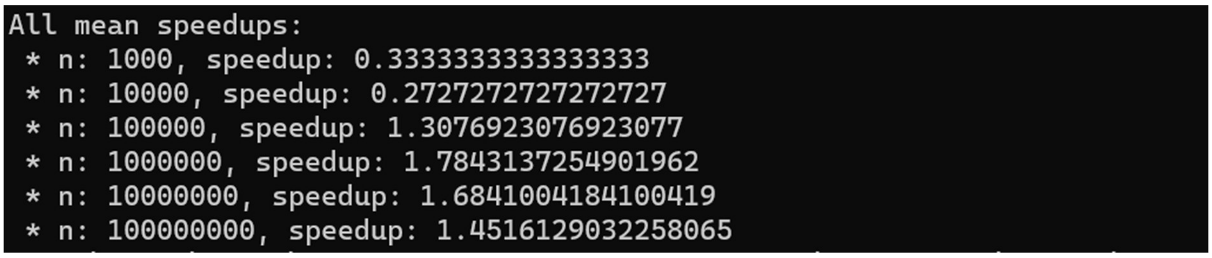
```
Running with n = 100000000:
  ...generating data...
  ...data done!

 * time: 7498.0 ms (sequential version)
 * time: 7323.0 ms (sequential version)
 * time: 7198.0 ms (sequential version)
 * time: 7202.0 ms (sequential version)
 * time: 7236.0 ms (sequential version)
 * time: 7238.0 ms (sequential version)
 * time: 7200.0 ms (sequential version)
 * time: 7181.0 ms (sequential version)
 * time: 7209.0 ms (sequential version)
 * time: 7435.0 ms (sequential version)
 * time: 7731.0 ms (sequential version)
 * time: 7352.0 ms (sequential version)
 * time: 7173.0 ms (sequential version)
 * time: 7197.0 ms (sequential version)
 * time: 7156.0 ms (sequential version)
 * time: 5132.0 ms (parallel version)
 * time: 5169.0 ms (parallel version)
 * time: 5043.0 ms (parallel version)
 * time: 4914.0 ms (parallel version)
 * time: 5088.0 ms (parallel version)
 * time: 4977.0 ms (parallel version)
 * time: 4960.0 ms (parallel version)
 * time: 4958.0 ms (parallel version)
 * time: 4893.0 ms (parallel version)
 * time: 4984.0 ms (parallel version)
 * time: 4915.0 ms (parallel version)
 * time: 5060.0 ms (parallel version)
 * time: 5022.0 ms (parallel version)
 * time: 4976.0 ms (parallel version)
 * time: 5064.0 ms (parallel version)
```

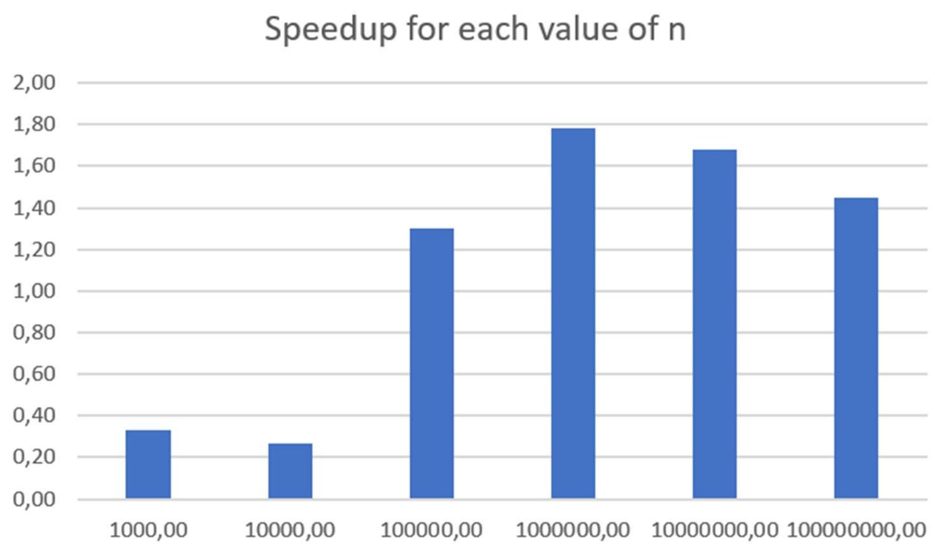


```
Running with n = 10000000:
  ...generating data...
  ...data done!

 * time: 820.0 ms (sequential version)
 * time: 804.0 ms (sequential version)
 * time: 805.0 ms (sequential version)
 * time: 779.0 ms (sequential version)
 * time: 807.0 ms (sequential version)
 * time: 776.0 ms (sequential version)
 * time: 760.0 ms (sequential version)
 * time: 753.0 ms (sequential version)
 * time: 739.0 ms (sequential version)
 * time: 770.0 ms (sequential version)
 * time: 906.0 ms (sequential version)
 * time: 972.0 ms (sequential version)
 * time: 867.0 ms (sequential version)
 * time: 887.0 ms (sequential version)
 * time: 817.0 ms (sequential version)
 * time: 465.0 ms (parallel version)
 * time: 486.0 ms (parallel version)
 * time: 478.0 ms (parallel version)
 * time: 477.0 ms (parallel version)
 * time: 478.0 ms (parallel version)
 * time: 475.0 ms (parallel version)
 * time: 477.0 ms (parallel version)
 * time: 479.0 ms (parallel version)
 * time: 484.0 ms (parallel version)
 * time: 470.0 ms (parallel version)
 * time: 517.0 ms (parallel version)
 * time: 536.0 ms (parallel version)
 * time: 497.0 ms (parallel version)
 * time: 475.0 ms (parallel version)
 * time: 479.0 ms (parallel version)
```



```
All mean speedups:
 * n: 1000, speedup: 0.3333333333333333
 * n: 10000, speedup: 0.2727272727272727
 * n: 100000, speedup: 1.3076923076923077
 * n: 1000000, speedup: 1.7843137254901962
 * n: 10000000, speedup: 1.6841004184100419
 * n: 100000000, speedup: 1.4516129032258065
```



Conclusion

In conclusion, from my testing I managed to get a speedup on all values starting at 100 000