IN4330 mandatory 4

Erlend Kristensen

28.04.2022

1 Introduction

This report is about implementing an algorithm for finding the convex hull around a set of random points. We will use a recursive algorithm, which works by drawing a line between the two outer most points (x-min and x-max), and finding the point furthest away from this line. Then repeat this process with a line between new point, leftmost point and new point, rightmost point, until there are no more points to find. We also have to include the points that are on the convex hull lines This is also done recursively. It starts when we do the normal recursive algorithm and find that the point furthest away is actually on the line. Then we recursively try to find if there are points between this new point, and the leftmost point and the rightmost point. The recursive logic works exactly the same as the algorithm described above.

We will also parallelize this algorithm to try to achieve some form of speedup. The challenge here is that the algorithm is a recursive one, so it is not easy to implement a parallel version.

2 User guide

Same user guide as found in Readme.txt file.

The code used to test is found in the folder "code". The benchmark folder is only used for benchmarking.
To compile:
For only serial part:
javac CH.java; java CH $< n >< seed >$

For parallel part:
javac CH_PARA.java; java CH_PARA $< n >< seed >< cores >$

Switch out $< n >< seed >< cores >$ with number you want.

n is number of points we want to use.
seed is the seed we want to use.
cores is number of cores we want to use.

if n $<$ 10 000, it will print to terminal, write to file and make a graph of the convex hull and all the points.
if cores is set to 0 or less, it will use the number of cores available on your computer.

The program runs 7 times in total, stores the time used for each run, then prints out the median time used.
In CH_PARA.java it does this for both parallel and sequential solution, and

also prints the speedup.

3 Parallel version

To make the parallel version, there are different approaches. I went with an approach of having a monitor keep track of the work available, and distribute this among the different threads. For each work done by a thread, the thread will keep one of the works itself, and send the other one to the monitor.
The only issue here is that I can't keep track of which order I should add the points to the convex hull, so I had to sort at the end of the algorithm. This sorting algorithm starts with either max-x or min-x (depending on which half circle its on), and finds the point closest to it within same half circle. Then it does the same with this new point, until there are no more points left.

For simplicity sake, i start with one half circle, then the next. It should be possible to achieve slightly better speedup if we do both at same time, but I did not implement that.

4 Implementation

As stated in the introduction, the algorithm is a sequential one. I chose to divide the task of doing the work above the first line (upper half circle) first, then the work bellow the first line. This made my algorithm more compact. When debugging, i used $n = 100$ and printed to terminal to see if the outputs were correct.
The program found in the folder "code" will do 7 runs and find the median

time used. Also, in the folder "myBenchmark" you can run a benchmarking test which tests both sequential and parallel solution. The only problem here is that it has to generate the points for each test, so the true difference won't be as noticable.

Good seeds to use for testing different cases were $99, 100, 101$.

5 Measurments

For all testing purposes of this report, the code was run on a laptop with a AMD Ryzen 5 5500U cpu with 6 cores and a base speed of 2.1 GHz.
The seed i used for time-testing was 101.

I did time tests and benchmarks for $n = 10$million twice in this report. Once before doing optimizations in my parallel code, and then after optimizations. The optimizations i did was to go from always sending work, to allowing the thread to keep one of the works. I also added a test that would allow a thread to stop sending work and doing rest sequentially after a certain amount of recursive depth was reached.

Here are the times (in seconds) before the optimization: before changes:
Speedup: 1.0487774602088573

Benchmark	Mode	Cnt	Score	Error	Units
MyBenchmark.testCH	thrpt	3	0,794	± 0,015	ops/s
MyBenchmark.testCH_para	thrpt	3	0,798	± 0,064	ops/s

After changes:

Median time Sequential: 0.4768412 Median time Parallel: 0.393558401 Speedup: 1.2116148423928574

Benchmark	Mode	Cnt	Score	Error	Units
MyBenchmark.testCH	thrpt	3	0,796	± 0,016	ops/s
MyBenchmark.testCH_para	thrpt	3	0,823	± 0,033	ops/s

As we can see, the stability and speed on the parallel solution got a lot better. As stated above, the benchmarking won't be very accurate due to us generating the points inside the benchmarking test, but the time test shows a speedup of about 20%, and the benchmarking clearly shows some speedup, so we can be happy with that!

Here is a graph of the times and speedups for $n = 100$ to $n = 10$million:

time in seconds my own laptop (6 cores)			
Number of points (n)	Time Sequential	Time Parallel	Speedup
100	1.959E-4	0.0086534	0.02263
1000	9.01399E-4	0.0092691	0.09724
10000	0.0029853	0.0134129	0.22256
100000	0.012643901	0.018577799	0.68059
1000000	0.1216863	0.0831292	1.46382
10000000	0.4768412	0.393558401	1.21161

6 Conclusion

In this report we have achieved an effective sequential algorithm for finding the convex hull around n random points. We have also parallelized this, and achieved a speedup of about 20%.