IN3030 Oblig 4

Marius Fredrichsen (mafredri)

Introduction

In this report I will compare a sequential and a parallel implementation of Convex Hull algorithm, testing with different values and how it was done.

User guide

To run the program simply run this in the terminal in the same folder as the files:

`Java ConvexHull -ea <n> <k>`

Note: n doesn't effect anything. I run the program for 10, 100, …, 100000000

Where n is the number of nodes and k is the number of threads. If k equals 0 it will take the k cores of the computer. Example:

Java Main -ea 2000000 0

The program will run 7 times and at the end give the median time of all the runnings and write it out in a csv file.

Convex Hull

The parallel version of Convex Hull was done by splitting up the data in k segments where k threads did the sequential algorithm on each part. This part didn't need any synchronizing except for .join() where after the threads were done we did the algorithm again on the resulting points from each thread. This way the sequential part at the end is done on a very small dataset of the full dataset. The points on the convex hull are included in the ending part since all points are processed.

Implementation

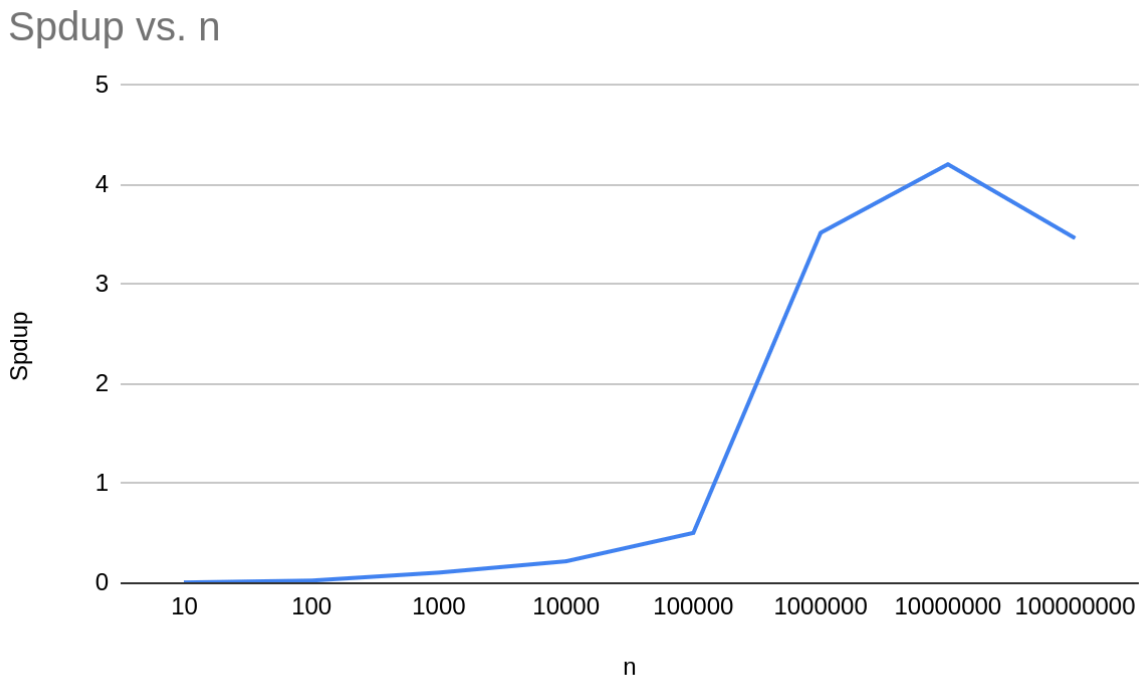
The algorithm consists of a recursive function that finds all the points from a point f to a point t. We run the algorithm on these two points and the third point which is the one furthest away from the line between t and f. Instead of checking every point in the algorithm we only

consider the ones that we have found to be on the right side. If there is a point that is on the line we do the algorithm again and check if the point is actually in between f and t.

Measurements

For the measurement I tested on a computer with 16 cores. The algorithms run 7 times in a for-loop where the times (ms) are stored and when done the median is used. Therefore the speedup below is based on median values of the time of running.

n	seq(ms)	par(ms)	Spdup
10	0,017	2,304	0,008
100	0,059	2,197	0,027
1000	0,449	4,199	0,107
10000	2,91	13,164	0,221
100000	7,262	14,362	0,506
1000000	69,069	19,618	3,521
10000000	653,503	155,26	4,209
100000000	6216,14	1793,614	3,466



In the graph / table above we see an increase of speedup when working with larger numbers, but a dip on very big inputs. The increase comes from the advantage of splitting up the data and working with a smaller part of the dataset later. The dip is kind of strange, but might be because the advantage of splitting up the data might not be that good when working with very large numbers. With 16 threads and 100000000 points, each thread works

with 6250000 points each. This might not be as efficient since the sequential algorithm halves the points for each recursive call. While the parallel version does that too, but not with that much less points.

Conclusion

We see an increase of speedup with larger numbers with the algorithm, but a small dip with very large numbers. This might be because of the way the algorithm halves each points for each recursive call.

Appendix

Its in Devilry.

Computer Specifications

Ram: 15,4 GiB Ram, LPDDR5, 6400 MT/s
CPU: 16 × 13th Gen Intel® Core™ i5-13500H