Oblig4 Report

Introduction

This report includes a user guide to my program. As well as an explanation behind my idea when making the convex hull algorithm in parallel, and how I have implemented both the sequential method and the parallel one. Then results and measurements of the times and speedup of the methods are presented, as well as discussion around this.

User guide – how to run my program

First compile the program using javac *java

To run the program write: java ConvexHull <n> <s> <met>
<n>: how many points to generate (must be greater than 2)
<s>: seed number for point generator
<met>: what method that will produce the hull-point output file (sequential or parallel). The sequential is default, but if you want parallel you can write "par" here.
This is not a required parameter but can be used if needed.

An example can be: java Convex 100 2 par
Or: java Convex 1000 3

Parallel Convex Hull

To parallel the method I used the tips section in the requirements.
My parallel algorithm consists of two methods, parMethod() and parSec(). The first does as the sequential, finds the points with min x and max x, creates a line and finds the points furthest away. After that it starts two threads that both call on parSeq with the new points. In the parSec() method everything is also as the sequential other than it (based on the level in the recursion calls) starts up two new threads for the right and left line of the point. New threads are created until a chosen level is achieved where the method instead calls the sequential method.

Implementation

Method ConvexHull(int n, int s):
This constructor sets the global variables. Then creates the x and y int arrays, then uses NPunkter17 to fill those arrays with point coordinates. Then MAX_X and MAX_Y is found when looking through the whole arrays, this is for using when drawing the graph later.

Method seqMethod():
Creates a empty IntList, koList. Then finds point p1 and p2 (point with max x value and point with min x value). Adds p1 to the koList.
Then finds the point furthest from the line p1-p2 on both sides, and split the point array into the points to the right of the line and points to the left of the line.

Then it calls the recursive method with the new furthest right point, and the IntList of all point on the right. Adds p2 to the koList. Then calls the recursive method with the furthest left point and the IntList of all the left points.
Then it returns the koList.

Method secReq (int p1, int p2, int p3, IntList m, IntList koList):
checks if there is only one in m and if p3 is this point, then p3 is added directly and the method returns.
If not the two new lines p1-p3 and p3-p2 are defined, then it finds the point with the max distance on the right (negative distance) from both the lines, as well as splitting the points into to IntList, one for points to the right of p1-p3 and one for points to the right of p3-p2. When doing this it also checks to see if the points are on the line (if the distance is 0), if it is the point is only added to the IntList and checked if it is the point furthest away, if the point has a x value and y value in-between the two points of the line. If it doesn't its not added.
After this, if there are point to the right of (or on) p1-p3, it calls the recursive method with the new furthest right point, and the IntList of all the points on the right of p1-p3. Adds p3 to the koList. Then, if there are points to the right of (or on) p3-p2, it calls the recursive method with the furthest left point and the IntList of all the points on the right of p3-p2.

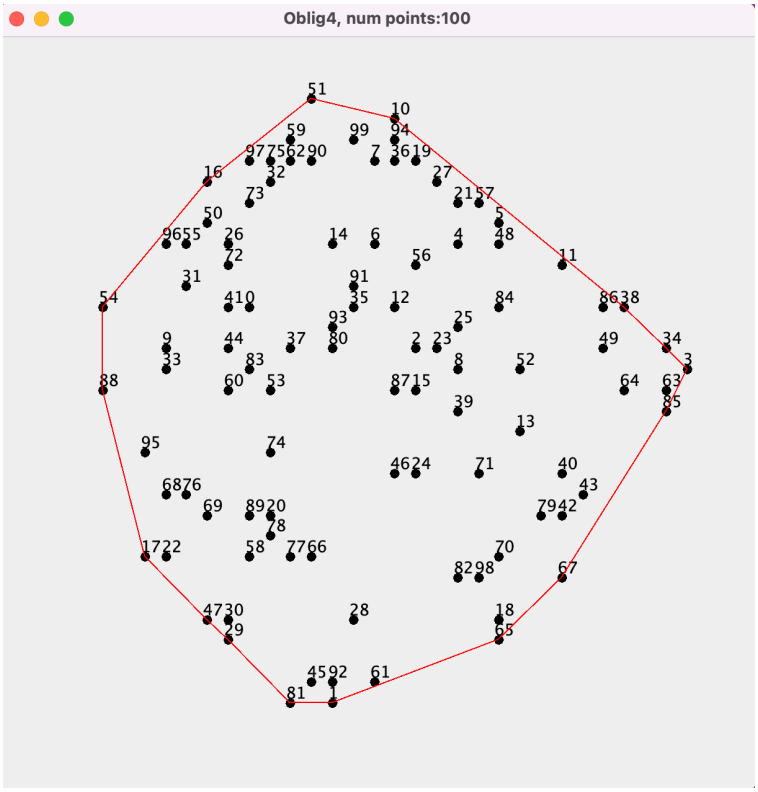
Method parMethod()
This method is the same as seqMethod(), the only change is that instead of calling the method seqRec(), the method creates two threads and two new IntList.
Then the two threads are started with one calling the method parSec() with the new furthest right point, and the IntList of all point on the right, as well as one of the new IntList and a int level with value 2 And the other thread calling parSeq()with the furthest left point and the IntList of all the left points, as well as one of the new IntList and a int level with value 2. After this it waits until both threads are done before adding p1 to the koList, then appending the right koList, p2 and the left koList. Then returning koList.

Method parRec(int p1, int p2, int p3, IntList m, IntList koList, int level)
This method is the same as seqRec(), the only change is before calling the method seqRec() it first check if the variable 'level' is under 4. If this is true it does not call seqRec() as usual, but instead creates two new threads and IntLists. Then the two threads are started with one calling, if there are any points to the right of (or on) p1-p3, the method parSeq() with the new furthest point, and the IntList of all points on the right of p1-p3. As well as one of the new IntList and a int level with value 2. And the other thread calling parSeq(), if there are any points to the right of (or on) p3-p2 with the furthest point and the IntList of all the points to the right of p3-p2, as well as one of the new IntList and a int level with value 2.
After this it waits until both threads are done before appending the right koList, p3 and the left koList. Then it lets the first thread call the parSec() method with the point furthest from the line p1-p3 and the IntList of all the points on the right of p1-p3, as well

as one of the new IntList and a int level with value 2. Then the other thread calls the parSec() method with the point furthest from the line p3-p2 and the IntList of all the points on the right of p3-p2, as well as one of the new IntList and a int level, 2.

Method main()
Reads the arguments for n, s and met.
Creates a ConvexHull with n and s.
Takes the time of the sequential method, and the parallel method. As well as calculating the speedup time. Then printing out both times + speedup.
Then Oblig4Precode is used to write hull points to a text file. And draw the two graphs.

Testing
To test that everything was working I have used drawGraph() and writeHullPoints() from the precode. With that I have checked to see that the algorithms do indeed create a convex hull, as well as checking that the written hull points is representative of what I saw on the drawing (all points on the straight lines on the edge are all recorded as hull points).



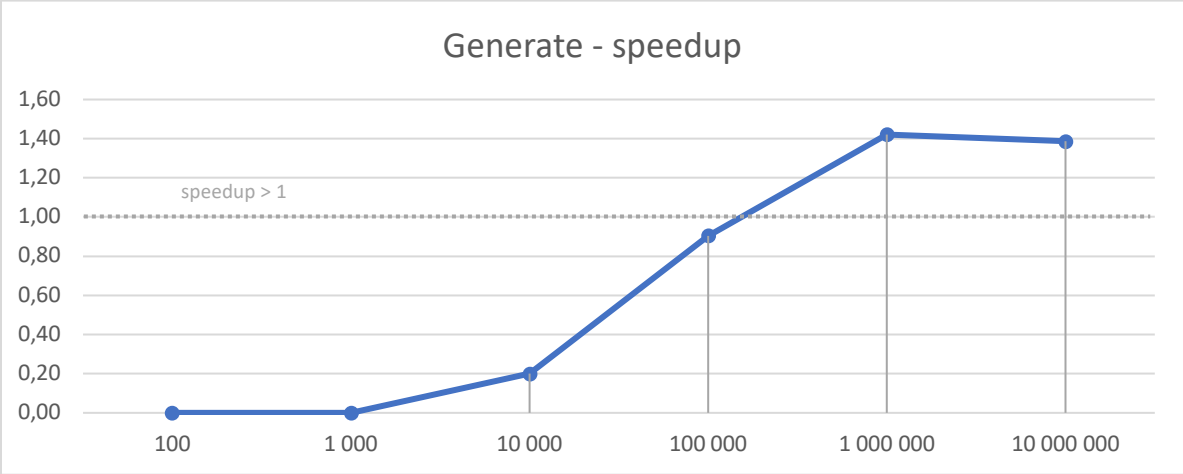
Measurements

To measure my results of times and speedup I have run my code with 6 different values of N: 100, 1000, 10.000, 100.000, 1.000.000, 10.000.000. I have run this on a computer with an Apple M1 Pro chip. It has 8 cores with a speed of 3.20 GHz, and with a main memory of 16GB RAM.

Below is a table of times, ms, for the sequential and parallel method, as well as speedup.

N- values	Seq	Par	Speedup
100	0	8	0,00
1 000	0	8	0,00
10 000	2	10	0,20
100 000	19	21	0,90
1 000 000	108	76	1,42
10 000 000	326	235	1,39

The times recorded are quite low even for large values of n. I believe this is because of how the algorithm works (finding the point furthest away), it will in many cases eliminate a lot of the points fast and only send a small subset of the points with the next call, which will make the time of that call way faster.



As seen speedup happens first when n = 1000000. The reason speedup does not happen on the lower n values is firstly that creating the threads creates some overhead that will increase the time, which will be noticeable for the lower n-s. We can also observe that the speedup does not increase from n = 1 mil to n = 10 mil, it actually goes down for the times I have recorded (this won't always be the case). It would be reasonable to think that with 8 threads doing the work sequentially in parallel I would get a speedup of 8, but I believe the speedup does not increase and go much over 1 because of how the points will be divided between threads. For many datasets the points will not be equally divided at all, and one thread may perhaps end up having a lot more points to sort through than others. Because the work is not evenly split between the threads it decreases the effect of the threads on the time. If only one thread has most of the points for example or the two threads in the start do most of the work, it will essentially work like a sequential method.

Conclusion

I have achieved implementing both a recursive sequential method and a recursive parallel method of the Convex Hull algorithm. I have achieved low run times for both methods, and a speedup > 1 when n = 1 mil and higher.