IN3030 – oblig 5

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Daniel Sharifi – danish

Contents

1. Introduction

This report is about parallelization of convex enveloping of a set of points, a recursive geometric problem. It discusses and compares the sequential and parallel version of the algorithms, and aims to see if and what benefits there are to gain by parallelizing the enveloping.

2. User guide

The main class is located in Main.java. To build the program compile all the java files with "javac *.java", and run it with one of the the following arguments:

(the first parameter specifies the mode)

Sequential algorithm:
java Main 0 {n}

Parallel algorithm:
java Main 1 {n} {threads}

Benchmarking (Sequential vs Parallel), this will run the parallel version with the specified thread count, and compare execution times with the sequential version for values of
n = {100, 1000, 10 000, 100 000, 1 mill, 10 mill}:
java Main 2 {threads}

3. Parallelization

All the threads first step is to find MIN_X and MAX_X which are minimum and maximum points with respect to their x coordinate, p1 being MIN_X and p2 MAX_X.

After this the threads draw a line from p1 to p2, and find the next two points, the points furthest away from the line on both sides of it.

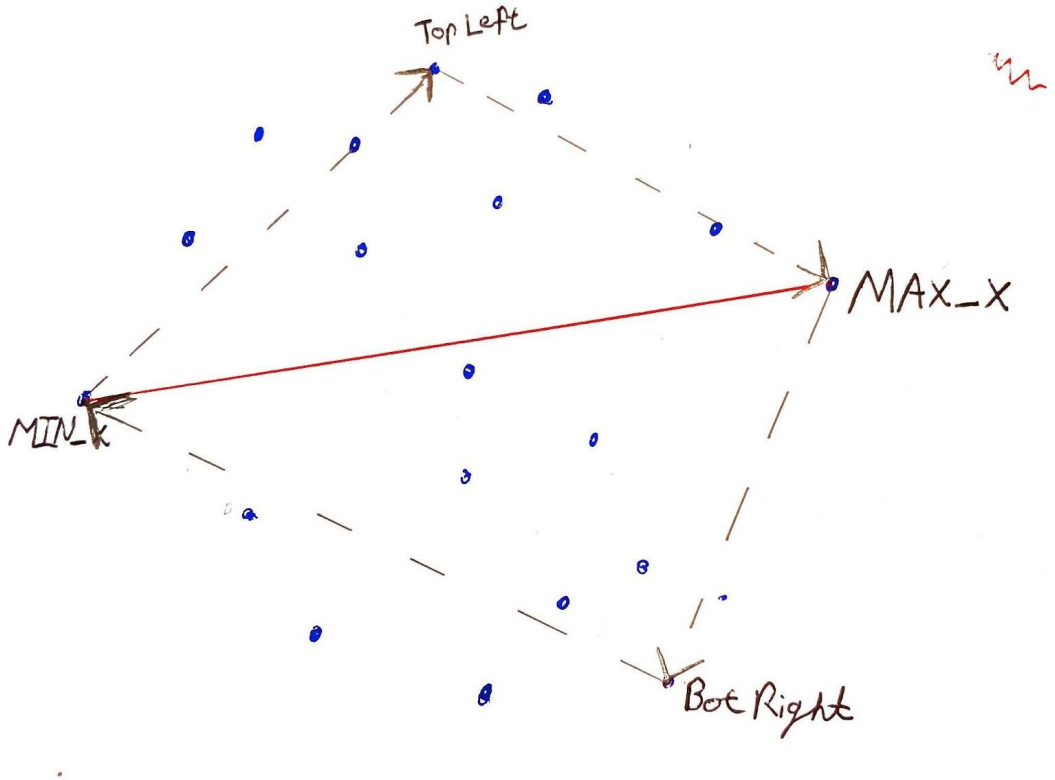


Figure1.

The algorithm has now found four points in the envelope, and will start 4 threads recursively finding values left of the line they are assigned.
Thread1 is assigned (TopLeft -> MAX_X), Thread2 (MIN_X -> TopLeft), etc.

In each recursion the thread will try to find the point furthest to the left of their assigned line. If it finds one, the thread will attempt to create a new thread that will do the right side of the recursion, and do the left side itself.

When there are no more points left outside a line, all other points on the same line will also be added to the binary tree, by first sorting them in order, then appending the intlist to the binary tree.

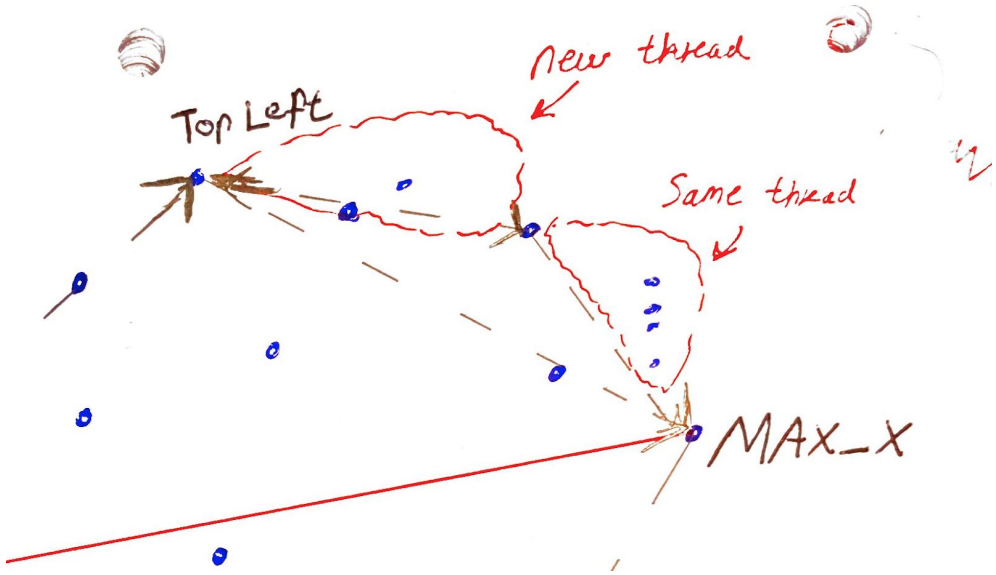


Figure 2.

Every furthest outer point for the line is added in a shared binary tree, before starting a new recursion/thread, which ensures it is thread safe.

4. Implementation

I have two worker classes (threads). The first class InitWorker is assigned to find the four points needed to create the four lines. While InitWorker is working, the main thread creates four new threads of Worker, which will do the recursion. When the InitWorker threads are all done, the Worker threads are immediately started.

The workers are assigned two points, which they will make a line of. Recursively the thread will find all points left of it, and find the one furthest away. It will then make two new lines from the original two points to the new point.

These two lines are then checked recursively. However, if there are fewer threads than the specified thread count, a new thread will be created to check the left side, as illustrated in figure 2. The thread count is implemented with an AtomicInteger.

```
// let new thread do leftSide;
if(threads == activeThreads.getAndIncrement()) {
new Thread(new Worker(p1, topLeft, leftSide, localEnvelope, path + "l")).start();
} else {
activeThreads.decrementAndGet();
recurse(p1, topLeft, leftSide, path + "l");
}
```

To minimize the number of synchronizations I've used a binary tree instead of a IntList to store each threads local envelope. To let each thread/recursion know where to put the point furthest, each recursion adds the character "r" (right) or "l' (left) depending on which side of the new point is recursed. This path is then used when adding the point to the binary tree.

All points between two edges are also included in the envelope. The points are added to an IntList as they're found. If no point outside left of the line is found, then this IntList is sorted, and at last appended to the binary tree.

Testing

To test that the values are correct for the parallelization i made a ToString() method for IntList and Oblig5 and printed out their found envelopes to debug and assert that it was correct.

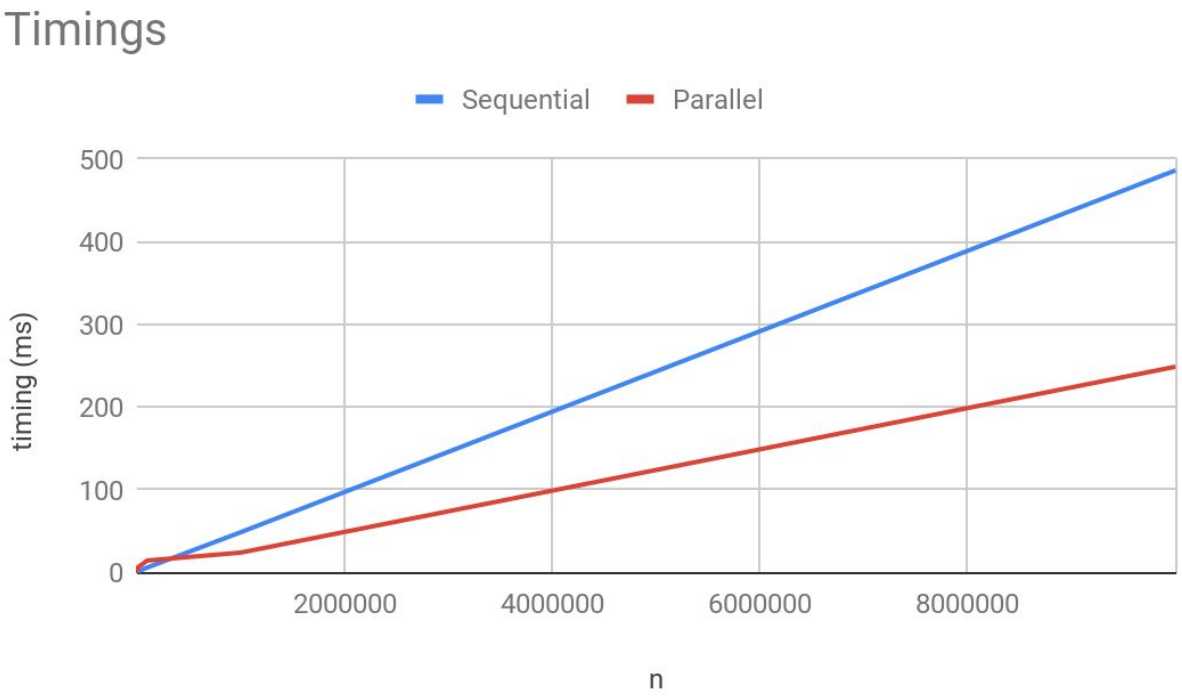
For testing, the benchmark mode is the desired mode to use.

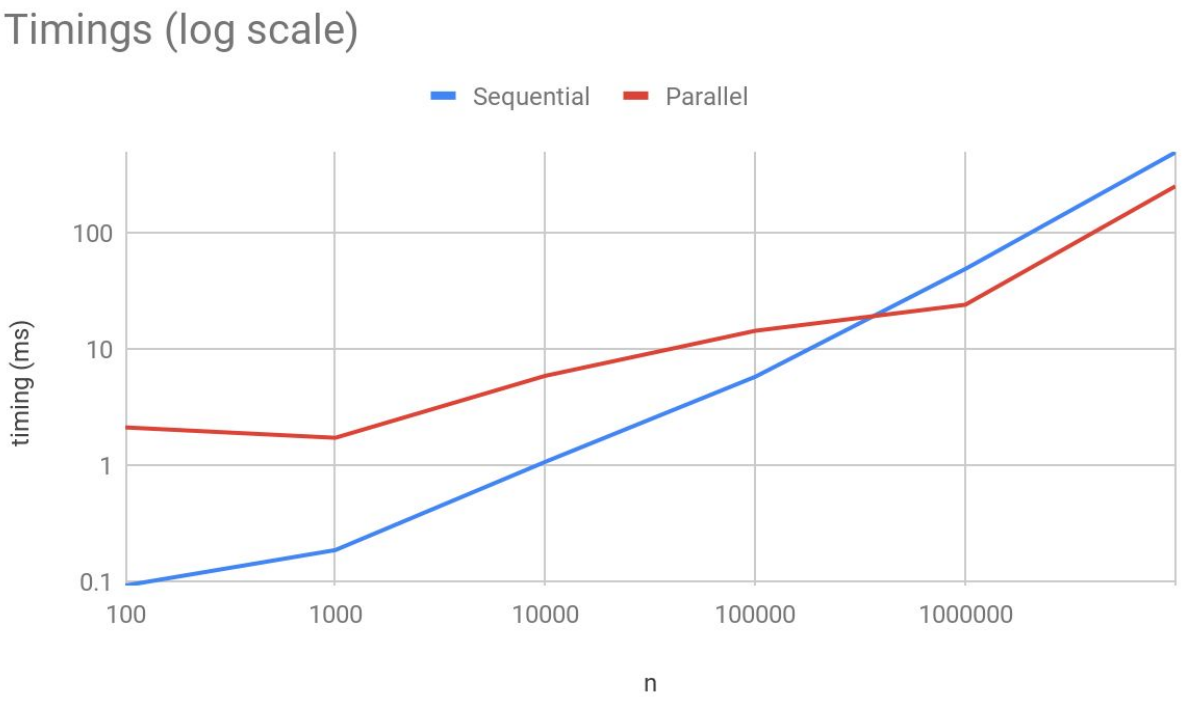
5. Measurements

All the timings are in milliseconds.

Sequential	Parallel	n (input size)
0.092515	2.109927	100
0.186346	1.718315	1000
1.067974	5.855753	10000
5.749218	14.268403	100000
48.475702	23.882051	1000000
486.315135	248.773967	10000000

Speedup	n (input size)
0.04384748856	100
0.1084469378	1000
0.1823803019	10000
0.4029335308	100000
2.029796436	1000000
1.95484737	10000000



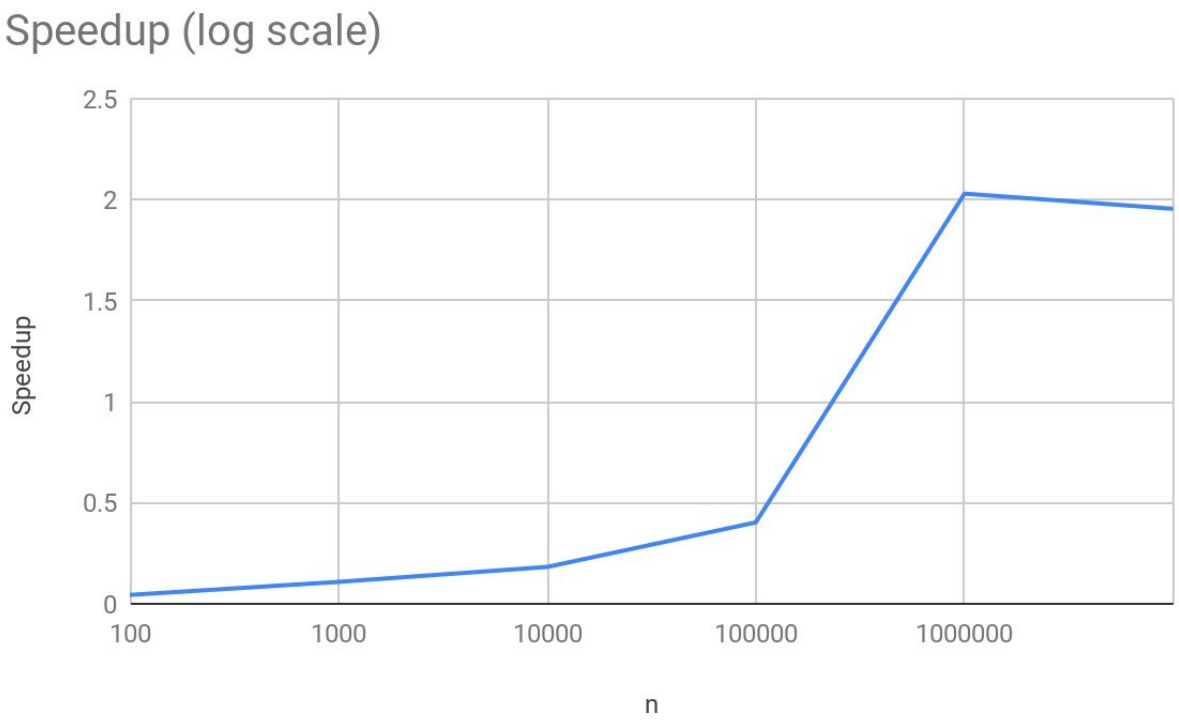


For the measurements I used my program's benchmarking mode using following arguments:

- java Main 2 8

Threads used = 8

This was run on an machine with 16gb ram and 4 cores (8 threads).

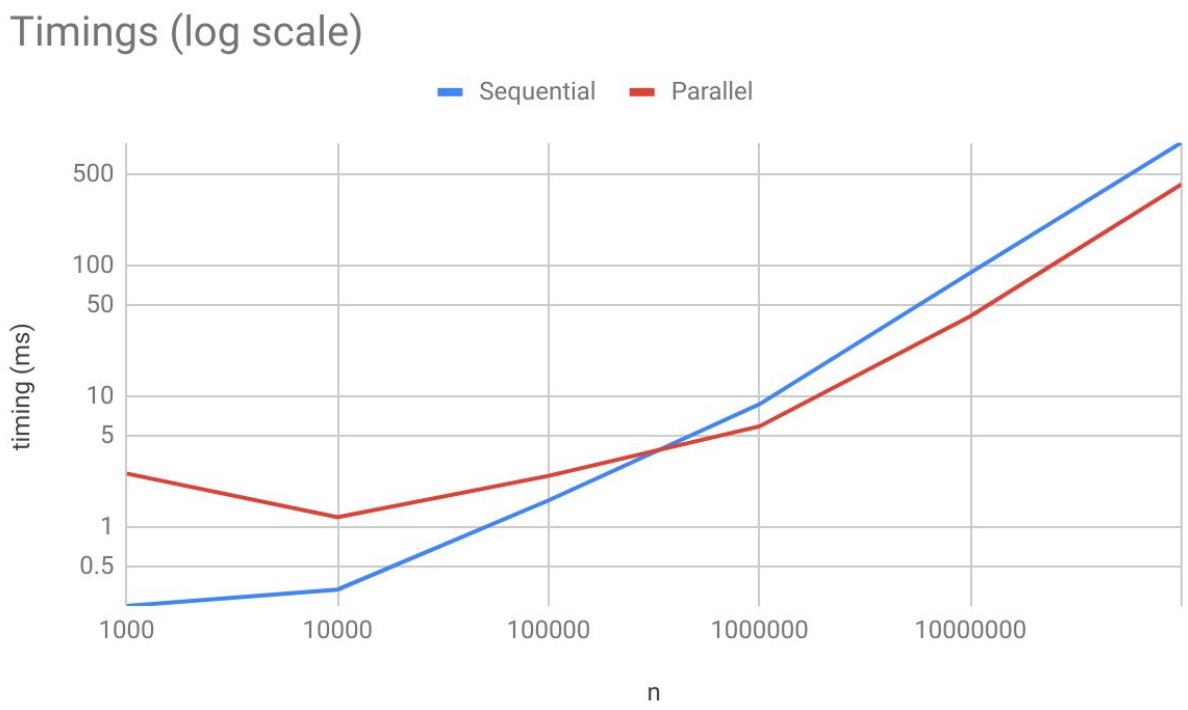


Looking at the speedup table we can conclude that using the parallelized algorithm is faster when n is equal to 1.000.000 or greater. The parallelization does not use IntLists, which are far more efficient than a binary tree in our case which gives a lot of overhead, each time creating a Node object and traversing the tree recursively to insert it. Thus the parallelization will get a speedup only once n grows significantly large enough, to allow for the binary tree cost being a smaller factor of the total time the algorithm is using.

One more factor to take into account is the garbage collector. Both algorithms use a lot of disposable objects such as IntLists for each recursion. The parallel algorithm also has two types of worker threads, where these threads are continually disposed when their life cycle is over. In addition the parallel algorithm has to start new worker threads continually for each recursion which is very cost heavy.

I believe I should have implemented a thread-pool to reuse the threads instead of disposing and creating new ones continually.

To compare the timing more easily I have included a graph of the timings with both axis using a log scale:



This shows that the parallel version is not increasing much in time for low values, because most of the time is spent on synchronizing instead of the “computation”, because of the high number of synchronizations.

In hindsight I believe the parallelization could have been done in a much more elegant and efficient matter. One of the problem with this algorithm is that it has the fewest number of threads for the first iteration of the recursion, and continually adds more threads for each recursion. When going into a new recursion the number of points to check is much lower, this means we have more threads when the points are fewer, and fewer threads when the dataset of points are greater, which is very unfortunate.

6. Conclusion

In conclusion we can see that the parallelized algorithm gives good speedup ratios about (2.0) when n = 1.000.000 or greater.

It could be even more efficient if a ThreadPool was implemented, and possibly avoiding a binary tree by only using IntLists and synchronizing each time to append them to each other.