Introduction

This report describes a multi-threaded Java implementation for finding the convex hull of a set of points, as well as presenting results and guides on how to use the program, under the assumption that the reader is using a Unix-like environment.

User Guide

The program uses the default package. Change working directory to the source folder, and run:

Figure 1: Compilation

```
$ javac *.java
```

The main function is located in the Main class. To run the program:

Figure 2: Running the program

```
$ java Main <max n> <runs> (debug option)
```

Where:
<Required> (Optional)

<max n>:	The maximum number of points to generate
<runs>:	The number of runs for each n
(debug option):	Either "v" (verbose printing) or "vd" (v + drawing)

Note: to test with same n as this report, more memory may be required by the Java VM. This can be allocated by passing the "-Xmx" flag, for instance: "-Xmx4G" allocates 4GB.

Parallel Algorithm

The sequential algorithm uses recursive functions that branch out in a recursion tree. The parallelization was done by exploiting that fact: by replacing recursive function calls with thread creation. Since thread creation is expensive, the algorithm falls back to recursive function calls after a limit is reached. In short, the algorithm creates a number of threads, where each thread performs the sequential algorithm in their subset of the problem.

Implementation

The function that starts the parallel algorithm is *parMethod()*. Instead of processing each half of the plane (containing the points) in two separate turns, parMethod() creates two threads of the Runnable class *ParEliminate* right away and starts the processing of the two halves in parallel. Important to note is that each thread gets their own local copy of an IntList to store their results (the points of the hull). The two lists are merged when both threads terminate. This ensures the points are stored in the correct order.

ParEliminate.run() is mutually recursive with *parRec()*, which is responsible for creating new threads and relieving control to the sequential algorithm when the depth limit is reached. Two threads branching right and left are created and started before calling join(). As in parMethod(), each thread gets their own, new copy of an IntList to store the results in. When join() returns, the IntLists of both threads are merged, which ensures the points are stored in the correct order.

Note that the depth limit is defined in the source code. Changing this value requires re-compilation of the program.

Measurements

The measurements are the median of 5 runs for every n using depth limit $DEPTH\_LIMIT = 4$.

Specifications:

- CPU: Intel Core i5-7300HQ @ 2.50GHz (4 cores)
- Memory: 8GB
- OS: 64bit Linux 4.15

n	sequential (ms)	parallel (ms)	speedup
100	0,07	1,61	0,04
1 000	0,15	1,11	0,13
10 000	1,05	4,96	0,21
100 000	4,75	5,47	0,87
1 000 000	48,73	25,93	1,88
10 000 000	527,54	264,07	2,00
100 000 000	4821,92	2328,07	2,07

Table 1: Sequential vs. Parallel results

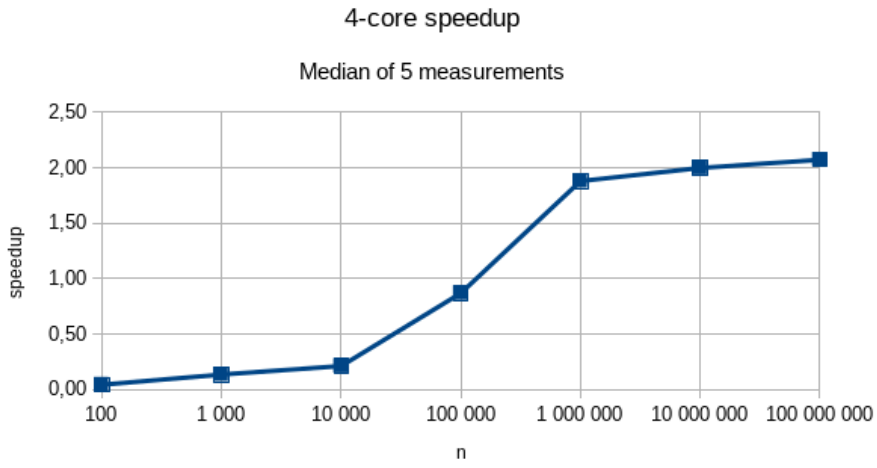

Figure 3: Speedup graph as function of n

We observe a slowdown for small n. This is expected, as the overhead of creating threads is large relative to the sequential execution time. A steady growth is observed from $n = 10^4$, and at $n = 10^7$ the execution time is halved compared to the sequential algorithm. The growth starts to halt from $n = 10^6$.

The parallel algorithm only loaded the CPU approximately 60 % during execution, meaning only (effectively) two of the four cores were utilized.

Conclusion

This implementation demonstrates that recursive algorithms can be parallelized using relatively small changes, by merely replacing recursive calls with thread creation and using trivial synchronization techniques.

Although speedups were achieved, the algorithm didn't fully utilize all cores on the test machine. Instead of only splitting the plane in half, the plane could be split vertically as well, dividing the problem into four sub-problems. This may improve CPU utilization and speedup when the number of cores is greater than 2.