Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 3
Quickhull

Trevor L. McDonell & Ivo Gabe de Wolff

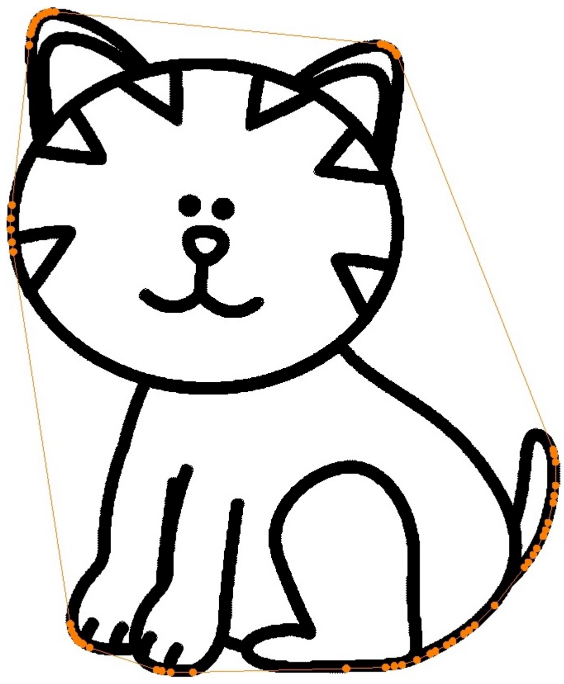
Deadline: Friday, 27 January, 23:59

Change Log

2023-01-09: Initial release

Introduction

In this assignment you will design and implement a data-parallel version of *Quickhull*, an algorithm to compute the smallest convex polygon containing a given set of points. A shape is *convex* if it does not have any dents. Formally, for any two points on the shape, the line between those shapes must also be fully contained in the shape.



As you might guess by its name, the Quickhull algorithm has some similarities with Quicksort: it is a divide-and-conquer algorithm which partitions the input data and recurses on these sub-partitions. In this practical, instead of using recursion to process the sub-partitions in task-parallel, you will implement this algorithm in data-parallel, where all segments are handled at once. To facilitate this you will use *Accelerate*, an embedded language in Haskell for data-parallel array computations that can run on the CPU or GPU. You can find more information on Accelerate from the lecture notes, the text book, and the library documentation.

Starting Framework

A few remarks regarding the starting framework:

- The starting framework can be found by accepting this assignment at the following GitHub Classrooms link: https://classroom.github.com/a/BVK-LdwR
- Installation instructions can be found in the file `README.md`
- The starting framework can (optionally) run using a Dev Container in Visual Studio Code. See the `README.md` for further instructions. Windows users in particular are *strongly encouraged* to use this. Alternatively, building in Windows under WSL2 is known to work, but building natively is currently only supported on *nix and MacOS.
- The starting template includes a small test suite to check your program, which you can run via `stack test`. To run a subset of the tests you can use the `--pattern` flag.
- The starting template includes a benchmark to gauge the performance of your solution and the speedup you achieve as you add more threads. These are the results I get from running my solution on my 4-core laptop (Intel i5-8259U):

```
$ stack run -- --benchmark --random 100000 --seed 1234
benchmarking quickhull
time                 6.920 ms   (6.679 ms .. 7.240 ms)
                     0.992 R²   (0.986 R² .. 0.998 R²)
mean                 9.471 ms   (8.862 ms .. 10.89 ms)
std dev              2.536 ms   (1.766 ms .. 3.672 ms)
variance introduced by outliers: 90% (severely inflated)
```

Data representation

The algorithm of the stating template uses a head-flags array to distinguish the different sections of the hull. It has the type:

```
type SegmentedPoints = (Vector Bool, Vector Point)
```

The two arrays are always of the same length. A flag value of `True` indicates that the corresponding point in the points vector is definitely on the convex hull. Let that point be called p_1. Let the next (left-to-right) point whose flag value is `True` be called p_2. We say that all of the points between p_1 and p_2 are in the same *segment*.

The elements whose head flag is `True` form the boundaries of the segments. These elements are known to be on the convex hull, and for the other elements the algorithm hasn't decided yet. The algorithm will proceed by repeatedly choosing one element per segment which is on the convex hull, removing elements which definitely do not belong to the convex hull, and partitioning the other elements to form new, smaller segments.

Initial partition

The main part of the algorithm always works on a segment of the array, formed by a line and the points on one side of that line. This is encoded in the type `SegmentedPoints` as described above. To start the algorithm, we must define the two initial segments by choosing a line (p_1, p_2) and splitting the remaining points into a set of points above this line, and a set of points below it. The output of this step be arrays of the form:

```
flags  = [True, False, False, …, True, False, False, …, True]
points = [p₁,   a₁,    a₂,    …, p₂,   b₁,    b₂,    …, p₁ ]
```

Where a_1, a_2, ..., are the points above the line (p_1, p_2), and b_1, b_2, ..., are those points below it. Placing the point p_1 again at the end will make parts of the rest of the implementation more convenient, but must be removed at the end. Some general hints:

- We must choose points p_1 and p_2 which are definitely on the convex hull.
- Points which lie *on* the line (p_1, p_2) should not be placed in either partition, as they are not part of the convex hull. One consequence of this is that the size of the output arrays can not be determined directly from the size of the input. Make sure that the middle point(s) along a line can not be chosen as the points p_1 or p_2, nor as the furthest point.
- Do not physically split the input into separate segments which are then concatenated, for example by using `filter` and `(++)`. This approach will not scale to the recursive step of the algorithm (more generally, consider how could you efficiently concatenate thousands of array segments?) and thus will earn you no marks.

Partition

After the initial partition is created, the `partition` step is executed repeatedly on the segmented representation until no undecided points remain. Each segment of the input consists of a different line segment. The process is similar to what was performed in the initial partitioning step, however now we need to perform the process over all line segments at once. Thus, instead of regular functions such as `scanl1` which operate over the entire array, we must use segmented versions which operate on each segment of a segmented array.

General remarks

- Make sure your program compiles using `stack build`.
- It is not allowed to concatenate arrays, for example with `(++)`. You must process all line segments at once in data-parallel!
- The various `scan*` and `permute` operators are very useful for this practical, pay attention to what they do.
- Include *useful* comments in your code. Do not paraphrase code but describe the structure of your program, special cases, preconditions, et cetera.
- Try to write readable and idiomatic Haskell. Style influences the grade! For example, use indentation of 2 spaces and a maximum column width of 80 characters. If you aren't sure, use a source code formatter such as ormolu:
  - https://hackage.haskell.org/package/ormolu
- Efficiency (speed) of your implementation influences the grade.
- Copying solutions—from other people, the internet, GitHub Copilot, or elsewhere—is not allowed.
- You cannot use external packages other than the dependencies which are already included in the template.
- Include your name(s) and student number(s) is the `README.md` file!

Submission

- This assignment may be submitted individually or in pairs.
- The deadline for this assignment is **Friday, 27 January, 23:59**. To submit your assignment, commit and push your changes to your GitHub Classrooms repository by the due date.
- For this assignment it is only necessary to edit file `src/Quickhull.hs` directory.
- Ensure that your submission compiles using the starting template.

1 (0.5 pt)**.** Implement the functions `shiftHeadFlagsL` and `shiftHeadFlagsR`, which given an array shifts the values one element to the left or right respectively.

2 (1 pt)**.** Implement the functions `segmentedScanl1` and `segmentedScanr1`, which are segmented variants of the inclusive scan operators `scanl1` and `scanr1`, respectively. These operators use a head flags array to indicate (by the value `True`) where each new segment should begin. You can assume that the first value in the flags array (left or right-most element, respectively) is `True`.

3 (0.5 pt)**.** Implement the functions `propagateL` and `propagateR`, which propagate (copy) the value whose corresponding head flag is `True` to respectively the *right* or the *left*, until another head flag with value `True` is encountered.

4 (3 pt)**.** Implement the function `initialPartition`.

5 (4 pt)**.** Implement the function `partition`.

6 (1 pt)**.** Finally, complete the algorithm by implementing the function `quickhull`.