Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 2
Delta Stepping

Trevor L. McDonell
Ivo Gabe de Wolff

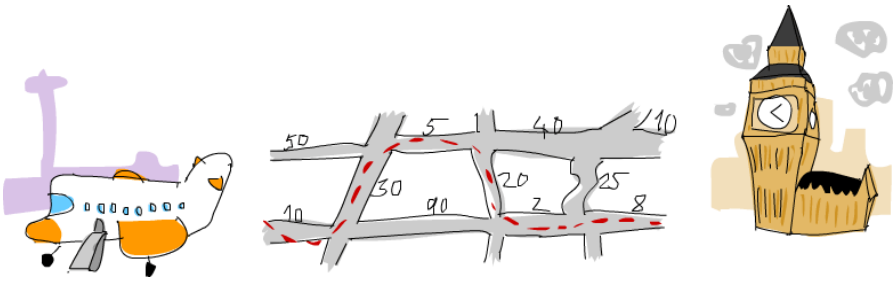
Deadline: Friday, 23 December, 23:59

Change Log

2022-12-05: Initial release

Introduction

Suppose your plane has just landed in England and that you need to drive from Heathrow airport to London as quickly as possible. There are a number of roads that you could take, which due to the current traffic each take a different amount of time to traverse. What would be the fastest way to reach your destination?



The problem of finding the shortest path between two points in a road map is a special case of a central problem in algorithmic graph theory: finding a path between two nodes (or vertices) in a graph such that the sum of the weights of its constituent edges is minimised.

In this practical assignment you will write a Haskell program to find the shortest distance from a given node to the other nodes in a graph. And, since this is a course about concurrency, naturally you will need to use multiple threads in order to compute this. To do this you will implement *Delta-Stepping*, a parallelisable single-source shortest path algorithm. More information about this algorithm can be found in the lecture notes.

Concurrency

You are free to use whatever techniques we have discussed in the course in order to implement this assignment. In particular, you must decide which parts of the algorithm *can* be parallelised, and then decide *how* to achieve this. You may implement a lock-based or lock-free solution, or use a combination of both techniques.

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/E1kPIHE-`
- Installation instructions can be found in the file `README.md`
- The starting framework uses two libraries in particular that you will need to interact with:
  - fgl: For querying the given graph structure
  - vector: For (mutable and immutable) arrays. The two flavours of vector that are particularly useful for this practical are:
    * `Data.Vector`: Boxed vectors which can store any structure, such as `Float`, `MVar`, `IntSet`, etc.
    * `Data.Vector.Storable`: Unboxed vector that can store only basic machine types, such as `Float`. However, you can get a pointer directly to the values using the function `unsafeWith`.

    As well as the mutable counterpart of each of these. You can convert between different (compatible) vector representations using the function `convert`.
- The template includes functions such as `forkThreads`, which provide a useful starting point for writing your own thread handling routines, as well as functions such as `printBucket` and `printCurrentState` to see the current state of the algorithm. The module `Sample.hs` contains a few example graphs, as well as the function `graphToDot` which renders a graph as an image.
- 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. While building the code in the regular Windows PowerShell should work without any problems, this configuration is not supported.
- 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 test --test-arguments='--pattern bench'
delta-stepping> test (suite: delta-stepping, args: --pattern bench)

All
  bench
    N1: OK (2.60s)
      67.9 ms ± 3.7 ms, 141 MB allocated, 254 KB copied,  96 MB peak memory
    N2: OK (1.26s)
      40.1 ms ± 3.7 ms, 141 MB allocated,  76 KB copied,  96 MB peak memory, 0.59x
    N4: OK (5.30s)
      20.4 ms ± 686  s, 141 MB allocated, 232 KB copied,  96 MB peak memory, 0.30x

All 3 tests passed (9.17s)
```

General remarks

Here are a few remarks:

- Make sure your program compiles using `stack`. Please do not submit attempts which don't even compile.
- It is recommended to first write a sequential implementation of the algorithm which you then parallelise. The starting template gives the structure of the basic sequential algorithm. You are free to change any part of this module other than the type of the `delta_stepping` function.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of the program, special cases, preconditions, et cetera.
- Try to write readable and idiomatic code. Style influences the grade! For example, use indentation of 2 spaces and a maximum column width of 80 characters. If in doubt, 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, or elsewhere—is not allowed.
- Include your name and student number in the `README.md` file!

Submission

- This assignment may be submitted individually or in pairs.
- The deadline for this assignment is **Friday, 23 December, 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 files in the `src` directory.
- Ensure that your submission compiles using the starting template.

1 (1 pt)**.** Implement the function `initialise`.

2 (3 pt)**.** Implement the function `findRequests`.

3 (3 pt)**.** Implement the function `relaxRequests`.

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