Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 2
Delta Stepping

Trevor L. McDonell
Ivo Gabe de Wolff
Tom Smeding

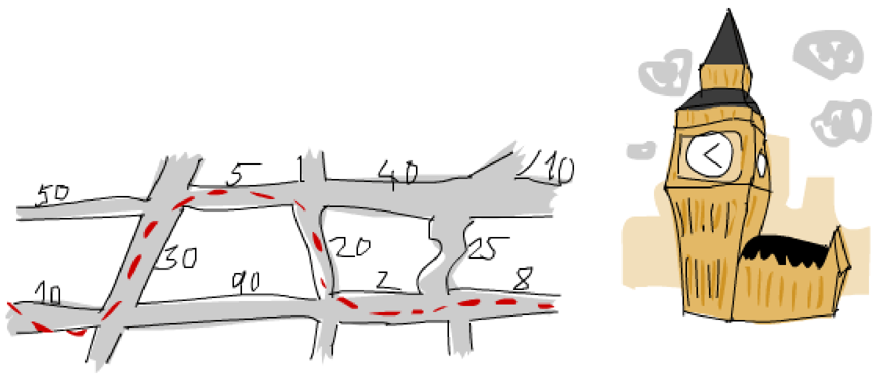
Deadline: Friday, 19 December, 23:59

Change Log

2024-11-26: Initial release

Introduction

Suppose your train has just arrived in England and that you need to get from St. Pancras to the Big Ben 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. We explain the algorithm in the *Delta stepping* lecture.

Concurrency

You are free to use whatever techniques we have discussed in the course in order to implement this assignment, as long as you attain the desired speedup in the benchmarks. In particular, you must decide which parts of the algorithm *should* 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:

- Find the starting framework on the website:
  `https://ics-websites.science.uu.nl/docs/vakken/b3cc/assessment.html`
  Remember to run `cabal update`, otherwise you might not have the required dependencies.

- The starting template contains various libraries that you will need to use. These are briefly discussed below.

- The template includes `forkThreads`, which provides a useful starting point for writing your own thread handling routines. The module `Sample.hs` contains a few example graphs.

- The main function of the algorithm, `deltaStepping`, takes a `Bool` as its first argument. This denotes whether the algorithm should run in a verbose mode. In the verbose mode, you may print any debug information to the console, which may be useful to debug your code.

- The starting template includes a small test suite to check your program, which you can run via `cabal run`. To run a subset of the tests you can use the `--pattern` flag, like `cabal run delta-stepping-test -- --pattern 'bench'`.

- 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 an M2 Max:

```
N1:       OK
  322  ms ±  13 ms, 715 MB allocated,  73 MB copied, 868 MB peak memory
N2:       OK
  181  ms ±  18 ms, 727 MB allocated,  67 MB copied, 868 MB peak memory, 0.56x
N4:       OK
  116  ms ± 9.4 ms, 756 MB allocated,  70 MB copied, 868 MB peak memory, 0.36x
N8:       OK
  96.2 ms ± 3.5 ms, 815 MB allocated,  62 MB copied, 868 MB peak memory, 0.30x
```

Dependencies

In this project, you will need to use several external modules that provide data structures to work with graphs or the internal state of your algorithm.

- `Data.Graph.Inductive.Graph` from fgl (Functional Graph Library)
  This library provides an interface to work with graphs. The functions from this library work on any type `gr` that implement a type class `Graph`. The template uses `Gr String Distance`, which implements that type class. You can thus use the functions from this library. Especially `order`, `labEdges` and `out` may be useful.

- `Data.IntSet` from containers
  This module provides an efficient data structure to store a set of integers. For the specific operations you need to do per bucket, this data structure will be more efficient than a simple list (`[Int]`).
  This module is imported under the name `IntSet`. You thus need to prefix any functions with `IntSet.`, e.g. `IntSet.empty` instead of `empty`.

- `Data.IntMap` from containers
  This module provides an efficient data structure for mappings from integers to certain values. You may also know this concept as a dictionary.
  This module is imported under the name `IntMap`. To split an `IntMap` in parts, you may want to use `splitRoot`, possibly multiple times.

- `Data.Vector.Mutable` from vector
  This module provides a vector data type, very similar to arrays. These vectors are mutable in IO blocks. The template uses these mutable vectors to store the array of buckets. It thus allows you to update buckets from IO blocks, possibly from multiple threads concurrently.
  This module is imported under the name `M`. You thus need to prefix any functions with `M.`. Note that an `IOVector` is an `MVector` instantiated such that it can be used in IO. All function from the documentation that work on an `MVector` can thus be used on an `IOVector`.

- `Data.Vector.Storable.Mutable` from vector
  This module provides *unboxed* vectors. These vectors directly contain values, instead of pointers to values, and are thus slightly more efficient than normal boxed vectors. You can however only use them with basic types, like `Float`. Hence the starting template uses boxed vectors for buckets, as they contain an `IntSet`, and an unboxed vector for the array of distances, as distances are floating-point numbers.

- `Utils` in the starting template
  Finally, the starting template contains some extra functions to work with these vectors. The function `atomicModifyIOVector :: V.IOVector a -> Int -> (a -> (a, b)) -> IO b` provides the functionality of `atomicModifyIORef` on a vector from `Data.Vector.Mutable`. The function `atomicModifyIOVectorFloat ::`

`M.IOVector Float -> Int -> (Float -> (Float, b)) -> IO b` provides this functionality on an unboxed vector (from `Data.Vector.Storable.Mutable`) of floating point numbers.

General remarks

- Make sure your program compiles using `cabal`. 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 `deltaStepping` 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. If you prefer an automatic formatter, you could use e.g. ormolu.[1]
- Efficiency (speed) of your implementation influences the grade.
- Copying solutions—from other people, the internet, or elsewhere—is not allowed. The AI Index[2] for this assignment is **1**: use of AI (LLMs) is *not* allowed.

Submission

- This assignment may be submitted individually or in pairs.
- The deadline for this assignment is **Friday, 19 December, 23:59**.
- Submission is via Brightspace; join one of the P2 groups and submit `src/DeltaStepping.hs` via Brightspace. As before, *only change* the file `src/DeltaStepping.hs`, otherwise the graders will not be able to compile your code.

Grading

1. (1 pt) Implement the function `initialise`.
2. (0.5 pt) Implement the function `allBucketsEmpty`.
3. (3 pt) Implement the function `findRequests`.
4. (3 pt) Implement the function `relaxRequests`.
5. (2.5 pt) Implement the function `step`.

[1] `https://hackage.haskell.org/package/ormolu`
[2] `https://www.uu.nl/sites/default/files/UU-AI-Index-student-EN.pdf`