Weekly Assignment 3
Data Parallel Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

November 2025

Introduction

This weekly assignment focuses on trees and pointer structures.

The handin is expected to consist of a report in either plain text or PDF file (the latter is recommended unless you know how to perform sensible line wrapping) of 4—6 pages, excluding any figures, along with an archive containing your source code. The report should contain instructions on how to run and benchmark your code.

Task 1: Tree Operations

This task was inspired by Eric Wastl.

Consider a tree described as a preorder traversal, through a sequence of *steps*. Each step is one of the following:

- `d`x: Go to the next child of the current node, which has an integral value of x.
- `u`: Go to the parent of the current node.

Steps are separated by whitespace. The first step must be `d`x. Examples can be seen on fig. 1. You can also use these to test your implementation. The first step must be `d`x. It is not allowed to add multiple root nodes.

You are given a code handout `trees.fut` with code that can parse a text representation of a traversal into an appropriate Futhark type. Your tasks for this assignment will be to decode the traversal into arrays that describe the depth and parent of each node in the tree, then to implement simple operations on trees. Note that the number of nodes in the tree is the same as the number of `d`x steps. The `u` steps to not contribute to the number of nodes in the tree.

Subtask 1.1: Implement `depths`

Implement the function `depths` that given an array of `steps` produces an array. The length should have an element for each node in the tree. Each element is of the form (d, v), where d is the depth of that node, and v is the value of that node (as given by the `d` step).

Analyse the work and span of your function. Is it work-efficient?

Hints:

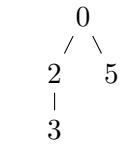
- You will need to implement your own function for computing exclusive scans.
- Remember that `u` steps do not add new nodes to the tree.

Subtask 1.2: Implement `parents`

Implement the function `parents` that given the D array produces the P array, as discussed in class.

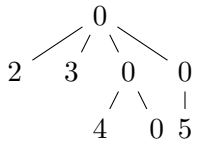
Analyse the work and span of your function. Is it work-efficient?

```
d0
 d2
  d3 u
  u
 d5
```



$$D = [0, 1, 2, 1]$$
$$V = [0, 2, 3, 5]$$
$$P = [0, 0, 1, 0]$$

```
d0
 d2 u
 d3 u
 d0
  d4 u
  d0 u
  u
 d0
  d5
```



$$D = [0, 1, 1, 1, 2, 2, 1, 2]$$
$$V = [0, 2, 3, 0, 4, 0, 0, 5]$$
$$P = [0, 0, 0, 0, 3, 3, 0, 6]$$

```
d1
 d2
  d3
   d4
    d5
```


$$D = [0, 1, 2, 3, 4]$$
$$V = [1, 2, 3, 4, 5]$$
$$P = [0, 0, 1, 2, 3]$$

Figure 1: Examples of trees and how they are encoded as steps. The indentation is solely for human readability. Note that we are not required to return to the root node at the end of the traversal.

Subtask 1.3: Implement `subtree_sizes`

A *subtree* rooted at i is the tree with node i as its root. The function `subtree_sizes` computes for each node i in a tree, the sum of all nodes of the subtree rooted at i. For example, for the tree

```
d0 d2 u d3 u d0 d4 u d0 u u d0 d5
```

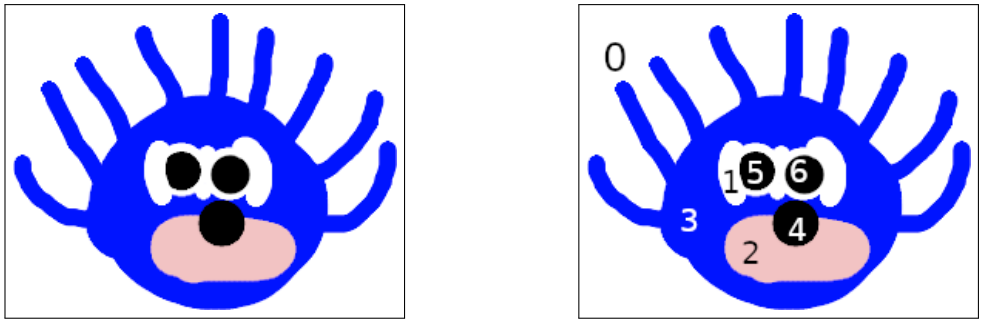
the result is

$$[14, 2, 3, 4, 4, 0, 5, 5]$$

Implement the function `subtree_sizes`. Analyse the work and span of your function. Is it work-efficient?

Hints:

- A simple implementation is bottom up iteration. If d is the maximum depth of the tree, start by considering nodes at depth d, then $d - 1$, etc.
- A more efficient implementation is possible by essentially doing a list ranking along every path from a leaf to the node. This is necessary to efficiently handle very tall and skinny trees. Note that in the degenerate case where each node in the tree has a single child, the problem becomes computing the (reverse) scan of a linked list.
- For the brave: your TA William believes that a particularly efficient implementation is possible by a solution that is inspired by the Blelloch scan algorithm. Talk to him if you are pretty sure the sun cannot melt those wax wings.



(a) Original artwork.

(b) Artwork with labeled regions.

Figure 2: Example of region labeling a piece of artwork. Note that unconnected areas with the same colour are considered distinct regions, and that the actual number used to label each region is arbitrary.

Task 2: Region Labeling

The *region labeling problem* is about uniquely labeling regions of an image. A *region* is a connected set of pixels that share the same colour. The idea can be generalised to structures that are not images, in which case it becomes a more general form of planer graph partitioning based on equivalence relationships between nodes. However, we will stick to images for simplicity. Figure 2 shows an example. The file `regionlabeling.fut` in the handout contains relevant definitions. It takes the form of a Literate Futhark program that can be run with

```
futhark literate regionlabeling.fut
```

to produce a Markdown and some images after region labeling. You do not need to use (or understand) the Literate Futhark directives in order to solve the assignment.

Subtask 2.1: Naive region labeling

We are given an an image represented by a two-dimensional array of colours `[h][w]u32` and must produce an array `[h][w]i64` that labels each pixel. A naive algorithm is as follows.

1. Label each pixel (i, j) as $i \times w + j$. Now each pixel has a unique label.
2. For each pixel, determine which of its four neighbours have the same colour, i.e., belong to the same region. Then update the label of each

pixel to be the maximum label of itself and the label of those of its neighbours that have the same colour.

3. Repeat step 2 until no more changes occur.

Implement this algorithm as a function

```
val region_label_naive [h] [w] : [h][w]u32 -> [h][w]i64
```

What is its work and span?

Subtask 2.2: Enlightened region labeling

In this subtask you will implement an algorithm for region labeling based on the research of your TA William Henrich Due. The idea is to construct a *directed acyclic graph* (DAG) with a node for each pixel in the image. An edge $a \to b$ denotes that a and b belong to the same region. An edge has *at most* one outgoing edge. If a node has no outgoing edges, we call it a *root*. A root can be seen as a *representative element* of a region, so we determine which region a node belongs to by following the path to the root. This means we do not care ultimately about which edges are present in the final DAG, only some path exists to the same root for two nodes that are in the same region.

We represent the DAG as a parent array D of type `[h*w]i64`. If $D[a] = b$, then that means D contains the edge $a \to b$. A root is identified by having itself as its parent.

The main challenge is to construct the DAG without introducing cycles. The key trick is that whenever we wish to insert an edge linking nodes a and b, we sort a and b by their original position in the grid, and create an edge from the smaller to the greater. The algorithm is as follows.

1. For each pixel a, for each of its (up to) four neighbours b, if a and b have the same colour, create an edge between a and b. The result is a one-dimensional array of desired edges.

2. Initialise the DAG D by having each node be its own parent, i.e., each node is initially a root.

3. Insert as many edges in D as possible. Use `reduce_by_index` to pick the largest target when conflicts occur. E.g. if we try to insert both edge $1 \to 2$ and $1 \to 3$, then it is the latter that "wins". Inserting an edge $a \to b$ is only possible when a is a root.

4. Keep the edges that were *not* inserted. This can be done with `filter` on the array of edges.

5. Update each un-inserted edge $a \to b$ to be $D[a] \to D[b]$.

6. If any un-inserted edges remain, go back to step 3.

7. Finally, for each element in the DAG compute the root, which can be done either naively or as a list-ranking-like operation, and turn it into a two dimensional array `[h][w]i64`.

Implement this algorithm as a function

```
val region_label_enlightened [h] [w] : [h][w]u32 -> [h][w]i64
```

What is its work and span? (Note: Deriving it fully is tricky. Consider making some assumptions to simplify the task.)

You are also allowed to tweak the algorithm to make it more efficient. In particular, it is possible to continuously compress paths from nodes to roots instead of waiting until the final step.