Advanced Techniques for Data Parallel Programming

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

Deduplication

Expansion

Text operations

Region Labeling

The problem

- *Deduplication* is removing duplicate elements from an array.
- Sometimes we want to preserve original element order, but sometimes not.

Python-ish sequential solution

```
def dedup(xs):
  seen = {} # A set
  out = []  # A list
  for x in xs:
    if x not in seen:
      seen.add(x)
      out.append(x)
  return add
```

Can you think of a simple parallel solution?

Sorting and packing

`pack xs` keeps only those elements that differ from the following element (and also the first element).

```
def pack [n] (xs: [n]i32) =
  zip3 (iota n) xs (rotate (-1) xs)
  |> filter (\(i,x,y) -> i == 0 || x != y)
  |> map (.1)
```

```
> pack [3,0,0,1,1,1,2,2,3,1]
[3, 0, 1, 2, 3, 1]
```

And then

```
def dedup xs = pack (sort xs)
```

Works for any orderable type, but we can do better in special cases.

Deduplication by `scatter`

- Deduplicates k numbers in the range $[0, n-1]$.
- Efficient when k is much bigger than n.

```
def dedup_scatter [k] (n: i64) (xs: [k]i64) : []i64 =
  scatter (replicate n false) xs (replicate k true)
  |> zip (iota n)
  |> filter (.1)
  |> map (.0)
```

Deduplication by histogram

Idea is similar to `scatter` above, but instead of storing true/false, we store the largest index of an element that hits that bucket,

```
def dedup_hist [k] 't (n: i64) (is: [k]i64) (xs: [k]t) : []t =
  let H = hist i64.max i64.lowest n is (indices xs)
  in map2 (\i j -> H[i] == j) is (indices xs)
     |> zip xs
     |> filter (.1)
     |> map (.0)
```

Deduplication

Expansion

Text operations

Region Labeling

Dealing with Irregular Parallel Problems

- Problems that are inherently irregular cannot be dealt with automatically by Futhark, but the problems can often be dealt with using, for instance, a *flattening-by-expansion* technique.
  Examples:
  1. Converting an array of lines to an array of points.
  2. Converting an array of triangles (or circles) to an array of lines.
  3. Work-efficient sieve of Erastothenes.

- Some problems may be harder to flatten and may essentially require *full flattening* (as Cosmin taught you).

Data Parallel Flattening by Expansion

- Programming technique developed by Martin Elsman and described in the paper *Data-Parallel Flattening by Expansion* (ARRAY, 2019).
- Presentation here also based on Martin's slides, but updated for language changes to Futhark.

Flattening tool: Segmented Iota

```
def segmented_iota [n] (flags:[n]bool) : [n]i64 =
  let iotas = segmented_scan (+) 0 flags (replicate n 1)
  in map (\x -> x -1) iotas
```

Example

```
flags           =  0   0   1   0   0   0   1   0
replicate n 1   =  1   1   1   1   1   1   1   1
iotas           =  1   2   1   2   3   4   1   2
map (-1) iotas  =  0   1   0   1   2   3   0   1
```

Flattening tool: Replicated Iota

```
def replicated_iota [n] (reps:[n]i64) : []i64 =
  let offsets = scan (+) 0 reps
  let h = hist (+) 0 (i64.sum reps) offsets (replicate n 1)
  in scan (+) 0 h
```

Example

```
reps          =  2   3   1   1
offsets       =  2   5   6   7
i64.sum reps  =  7
h             =  0   0   1   0   0   1   1
scan (+) 0 h  =  0   0   1   1   1   2   3
```

The expand function

```
val expand 'a 'b :
      (sz: a -> i64)
  -> (get: a -> i64 -> b)
  -> []a
  -> []b
```

The expand function

```
val expand 'a 'b :
     (sz: a -> i64)
  -> (get: a -> i64 -> b)
  -> []a
  -> []b
```

Example

```
> expand (\x -> x) (\x i -> (x,i)) [0,1,2,3]
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
```

Definition of expand

```
def expand 'a 'b
           (sz: a -> i64) (get: a -> i64 -> b) (arr: []a) =
  let szs = map sz arr
  let idxs = replicated_iota szs
  let flags = map2 (!=) idxs (rotate (-1) idxs)
  let iotas = segmented_iota flags
  in map2 (\i j -> get arr[i] j) idxs iotas
```

Definition of expand

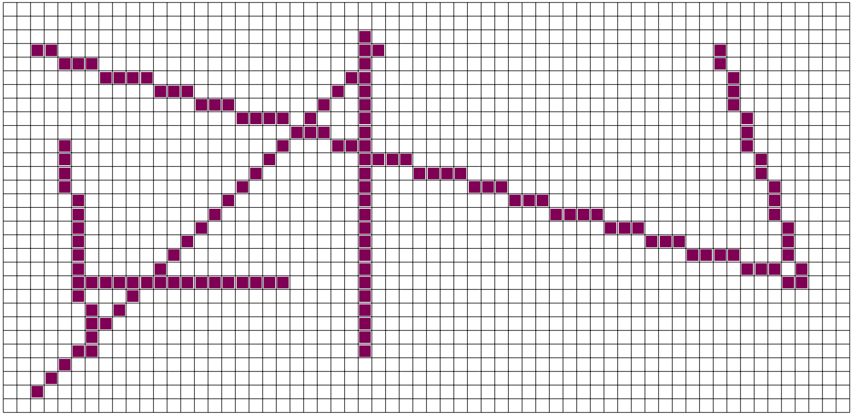
```
def expand 'a 'b
           (sz: a -> i64) (get: a -> i64 -> b) (arr: []a) =
  let szs = map sz arr
  let idxs = replicated_iota szs
  let flags = map2 (!=) idxs (rotate (-1) idxs)
  let iotas = segmented_iota flags
  in map2 (\i j -> get arr[i] j) idxs iotas
```

Example

```
 szs   | 0 1 2 3
 idxs  | 1 2 2 3 3 3
 flags | T T F T F F
 iotas | 0 0 1 0 1 2
```

Case study: line rasterisation

Given a sequence of $(start, end)$ pairs, draw the lines on a grid by interpolating between start and end positions.



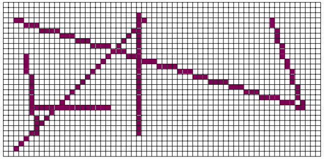
Main challenge: number of pixels per line depends on coordinates.

Drawing the grid

```
def mk_grid [n] (h:i64) (w:i64)
                (xys:[n](i64,i64)) : [h][w]u32 =
  let is = map (\(x, y) -> if 0<=y && y<h && 0<=x && x<w
                           then w*y+x else -1) xys
  let flatgrid = replicate (h*w) 0
  let ones = replicate n 1
  in unflatten (scatter flatgrid is ones)
```

Find Points in Parallel (I)

- Number of points on a line is the maximum of the *x*-range and the *y*-range.
- Longer lines, more work!



```
type point = (i64, i64)
type line = (point, point)

def compare (v1: i64) (v2: i64) : i64 =
  if v2 > v1 then 1 else if v1 > v2 then -1 else 0

def slope ((x1, y1): point) ((x2, y2): point) : f64 =
  if x2 == x1
  then if y2 > y1 then 1 else -1
  else f64.i64 (y2 - y1) / f64.i64 (i64.abs (x2 - x1))
```

Find Points in Parallel (II)

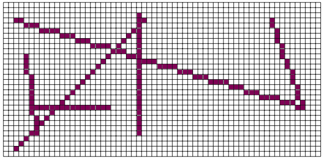
- Given a line, we can compute the number of points the line expands to!
- Given a point index, we can compute the point coordinates.

```
def points_in_line ((x1, y1), (x2, y2)) : i64 =
  1 + i64.max (i64.abs (x2 - x1)) (i64.abs (y2 - y1))

def get_point_in_line ((p1, p2): line) (i: i64) : point =
  if i64.abs (p1.0 - p2.0) > i64.abs (p1.1 - p2.1)
  then let dir = compare p1.0 p2.0
       let sl = slope p1 p2
       in ( p1.0 + dir * i
          , p1.1 + i64.f64 (f64.round (sl * f64.i64 i)))
  else let dir = compare (p1.1) (p2.1)
       let sl = slope (p1.1, p1.0) (p2.1, p2.0)
       in ( p1.0 + i64.f64 (f64.round (sl * f64.i64 i))
          , p1.1 + i * dir)
```

Find Points in Parallel (III)

We now use expand to convert an array of lines into an array of points.



```
def drawlines [n] (h: i64) (w: i64) (lines: [n]line) : [h][w]u32
  let xys =
    expand points_in_line get_point_in_line lines
  in mk_grid h w xys
```

Deduplication

Expansion

Text operations

Region Labeling

Characters and Strings

We consider a character to be simply a byte.

```
type char = u8
```

Characters and Strings

We consider a character to be simply a byte.

```
type char = u8
```

A string is an array of characters.

```
type string [n] = [n]char
```

String to integer

```
def is_digit (c: u8) = c >= '0' && c <= '9'
def dtoi (c: u8): i32 = i32.u8 c - '0'

def atoi [n] (s: string [n]) : i32 =
  ???
```

Any ideas?

String to integer (parallel)

```
def is_digit (c: u8) = c >= '0' && c <= '9'
def dtoi (c: u8): i32 = i32.u8 c - '0'

def atoi [n] (s: string [n]) : i32 =
  i32.sum (map2 (*)
                (map dtoi (reverse s))
                (tabulate n (\i -> 10 ** i32.i64 i)))
```

String to integer (sequential)

```
def is_digit (c: u8) = c >= '0' && c <= '9'
def dtoi (c: u8): i32 = i32.u8 c - '0'

def atoi [n] (s: string [n]) : i32 =
  let (sign, s) =
    if n > 0 && s[0] == '-'
    then (-1, drop 1 s)
    else (1, s)
  in sign
     * (loop (acc, i) = (0, 0)
        while i < length s do
          if is_digit s[i]
          then (acc * 10 + dtoi s[i], i + 1)
          else (acc, n)).0
```

Word splitting

```
ghci> words "data parallel programming"
["data","parallel","programming"]
```

How do we do this in Futhark?

1. How should we represent the result?
2. Can we do this in parallel?
3. Can we give it a safe interface?
    - Not strictly speaking a DPP topic.

Word splitting: Representation

We represent a **substring** as an *offset* and *length* into some other string.

- Having data that is relative to some "context" (the full string) is a common trick in languages like Futhark.
- In a language with pointers, a substring could just contain a pointer to the full string.

```
type slice = (i64, i64)

def get 't ((start, end): slice) (xs: []t) =
  xs[start:end]

> words "data parallel programming"
[(0, 4), (5, 13), (14, 25)]
```

Word splitting: working through it

Suppose `s : [n]i8 = "foo bar baz"`.

- `let flags = map is_space s`
  - `f f f t f f f t f f f`
- `let xs = map (\ c -> i64.bool (isnt_space)) s`
  - `1 1 1 0 1 1 1 0 1 1 1`

Word splitting: working through it

Suppose `s : [n]i8 = "foo bar baz"`.

- `let flags = map is_space s`
  - `f f f t f f f t f f f`
- `let xs = map (\ c -> i64.bool (isnt_space)) s`
  - `1 1 1 0 1 1 1 0 1 1 1`
- `let s’ = segmented_scan (+) 0 flags xs`
  - `1 2 3 0 1 2 3 0 1 2 3`

Word splitting: working through it

Suppose `s : [n]i8 = "foo bar baz"`.

- `let flags = map is_space s`
  - `f f f t f f f t f f f`
- `let xs = map (\ c -> i64.bool (isnt_space)) s`
  - `1 1 1 0 1 1 1 0 1 1 1`
- `let s’ = segmented_scan (+) 0 flags xs`
  - `1 2 3 0 1 2 3 0 1 2 3`
- `let triples = zip3 (iota n) s’ (rotate 1 s’)`
  - `0 1 2 3 4 5 6 7 8 9 10`
  - `1 2 3 0 1 2 3 0 1 2 3`
  - `2 3 0 1 2 3 0 1 2 3 1`

Word splitting: working through it

Suppose `s : [n]i8 = "foo bar baz"`.

- `let flags = map is_space s`
  - `f f f t f f f t f f f`
- `let xs = map (\ c -> i64.bool (isnt_space)) s`
  - `1 1 1 0 1 1 1 0 1 1 1`
- `let s' = segmented_scan (+) 0 flags xs`
  - `1 2 3 0 1 2 3 0 1 2 3`
- `let triples = zip3 (iota n) s' (rotate 1 s')`
  - `0 1 2 3 4 5 6 7 8 9 10`
  - `1 2 3 0 1 2 3 0 1 2 3`
  - `2 3 0 1 2 3 0 1 2 3 1`
- `let k = filter (\(i, x, y) -> x > y || (i == n - 1 && x > 0)) triples`
  - `[(2, 3, 0), (6, 3, 0), (10, 3, 1)]`

Word splitting: working through it

Suppose `s : [n]i8 = "foo bar baz"`.

- `let flags = map is_space s`
  - `f f f t f f f t f f f`
- `let xs = map (\ c -> i64.bool (isnt_space)) s`
  - `1 1 1 0 1 1 1 0 1 1 1`
- `let s’ = segmented_scan (+) 0 flags xs`
  - `1 2 3 0 1 2 3 0 1 2 3`
- `let triples = zip3 (iota n) s’ (rotate 1 s’)`
  - `0 1 2 3 4 5 6 7 8 9 10`
  - `1 2 3 0 1 2 3 0 1 2 3`
  - `2 3 0 1 2 3 0 1 2 3 1`
- `let k = filter (\(i, x, y) -> x > y || (i == n - 1 && x > 0)) triples`
  - `[(2, 3, 0), (6, 3, 0), (10, 3, 1)]`
- `map (\(i, x, y) -> (i - x + 1, i + 1)) k`
  - `[(0, 3), (4, 7), (8, 11)]`

Word splitting (the final function)

```
type slice = (i64, i64)

def words [n] (s: string [n]) : []slice =
  segmented_scan (+) 0
     (map is_space s) (map (isnt_space >-> i64.bool) s)
  |> (\x -> zip3 (indices s) x (rotate 1 x))
  |> filter (\(i, x, y) -> x > y || (i == n - 1 && x > 0))
  |> map (\(i, x, _) -> (i - x + 1, i + 1))
```

Arbitrary splitting

```
type slice = (i64, i64)

def break [n] 't (p: t -> bool) (s: [n]t) : []slice =
  segmented_scan (+) 0
     (map p s) (map (p >-> not >-> i64.bool) s)
  |> (\x -> zip3 (indices s) x (rotate 1 x))
  |> filter (\(i, x, y) -> x > y || (i == n - 1 && x > 0))
  |> map (\(i, x, _) -> (i - x + 1, i + 1))
```

Word splitting: type safety

It is a problem that we can use our "words" with the wrong context string.

```
> let w = head (words "foo bar baz")
> get w "data parallel programming"
" pa"
> get w "foo"
Index [4:7] out of bounds for array of shape [3].
```

Can we fix this?

A tool: existential sizes

Futhark is mostly a simple language, but it has (at least) **one weird type system feature**: *size types*.

```
val zip 'a 'b [n] : [n]a -> [n]b -> [n](a,b)
```

A tool: existential sizes

Futhark is mostly a simple language, but it has (at least) **one weird type system feature**: *size types*.

```
val zip 'a 'b [n] : [n]a -> [n]b -> [n](a,b)

val filter 'a [n] : (p: a -> bool) -> (as: [n]a) -> ?[d].[d]a
```

When we call a function with an *existential return type* we get back a size that is *distinct from every other size* (as far as the type checker is concerned).

```
> filter (>0) [1,-2,3]
[1, 3]
> filter (>0) [1,2,-3]
[1, 2]
> > zip (filter (>0) [1,-2,3]) (filter (>0) [1,2,-3])
Cannot apply "zip" to "(filter (> 0) [1, 2, -3])" (invalid type).
Expected: [n]i32
Actual:   [m]i32

Sizes "n" and "m" do not match.

Note: "n" is unknown size returned by "filter" at 1:6-26.

Note: "m" is unknown size returned by "filter" at 1:29-49.
```

Word splitting: using phantom sizes

Idea: use an abstract type and size to represent a word, and also return a function for extracting an array from the word, rather than having a general `get` function.

```
type word [p]

val words [n] : [n]char -> ?[p][k].(word [p] -> ?[m].[m]char,
                                    [k](word [p]))
```

Here `p` *is not the size of anything specific*, but a "phantom size" where all we know is that it is different from all other sizes.

```
> let (get, ws) = words "foo bar baz" in get ws[0]
[102, 111, 111]
```

Word splitting: using phantom sizes

Idea: use an abstract type and size to represent a word, and also return a function for extracting an array from the word, rather than having a general `get` function.

```
type word [p]

val words [n] : [n]char -> ?[p][k].(word [p] -> ?[m].[m]char,
                                    [k](word [p]))
```

Here `p` *is not the size of anything specific*, but a "phantom size" where all we know is that it is different from all other sizes.

```
> let (get, ws) = words "foo bar baz" in get ws[0]
[102, 111, 111]
```

```
> let (get0, ws0) = words "foo bar baz"
  let (get1, ws1) = words "another"
  in get1 ws0[0]
Cannot apply "get1" to "ws0[0]" (invalid type).
Expected: word [p1]
Actual:   word [p0]
```

Implementation

Splitting logic the same as before, and the phantom size p can be picked arbitrarily.

```
type word [p] = ([p](), i64, i64)

def words [n] (s: [n]char) =
  ( \(_, i, k) -> s[i:i + k]
  , ...
    |> map (\(i, (x, _)) -> ([], i - x + 1, x))
```

It is not a core part of DPP, but it is an interesting research problem to see how types can be used to make data parallel programming safer.

Deduplication

Expansion

Text operations

Region Labeling

Computing transitive equality relations

Consider that we have a set

$$\{\mathcal{X} = x_0, x_1, \ldots, x_{n-1}\}$$

and a relation $E : \mathcal{X} \times \mathcal{X}$ denoting partial equalities on elements of $\mathcal{X}$.

- Not a full equality relation - it may be that $(x, y) \in E$, $(y, z) \in$, $(x, z) \notin E$.
- But we do reflexivity such that $(x, y) \in E \Rightarrow (y, x) \in E$.

Now suppose we want to build a proper transitive equality relation E^* such that

$$(x, y) \in E^*$$

iff there is a set

$$\{(x, x_1), \ldots, (x_i, y)\} \subset E$$

A graph perspective

Another way of looking at this problem is saying that we have a collection of vertices

$$V = \{\mathcal{X} = x_0, x_1, \ldots, x_{n-1}\}$$

and a collection of (undirected) edges

$$E : \mathcal{X} \times \mathcal{X}$$

and we want to find all the *connected components*.
Ultimately, given two nodes, we want to quickly determine whether there is a path from one to the other.

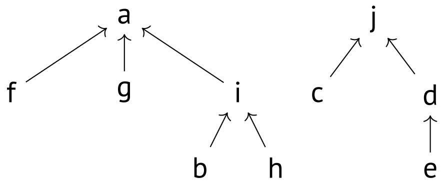
Equivalence classes

- Basically, given **partial information** about equalities, split a set into **equivalence classes**.
- Each equivalence class has a **representative element** that identifies the class.
- Useful building block for various algorithms, including type checking, constraint solving, and the DPP third weekly assignment...

Let us look at how it can be done in parallel.

Equivalence classes as forests

- **Basic idea:** represent each equivalence class as a *directed* tree, with edges going towards the representative element.



This represents the equivalence classes $\{a, f, g, i, b, h\}$ and $\{b, c, d, e\}$.

- To determine whether two elements are equal, follow the paths to their roots and see if they have the same representative element.
  - We write this as $r(x)$ (for "representative").
- Precise structure of the trees is *not important*—we will exploit this to construct them efficiently in parallel.

Constructing equivalence class trees sequentially

1. We are given the set $\mathcal{X}$ of nodes and a list E of undirected edges.
2. Initially, create a forest with a vertex for each element of $\mathcal{X}$ and **no edges**.
3. As long as there are edges left in E:
    3.1 Remove an edge (x_i, x_j) from E and add $(r(x_i), x_j)$ to the forest.

The resulting forest may contain unbalanced trees, but that is not important for correctness.

Example

Consider

$$\mathcal{X} = \{a, b, c, d\}$$

and

$$E = \{(a, b), (a, c)\}$$

Going through example on whiteboard...

Parallel construction

- We represent the forest by its parent vector, and a node is an index.

```
-- Find representative element.
def r (x: i64) (forest: []i64) : i64 =
  loop x = x while forest[x] != r do forest[x]
```

Main thing to parallelise is edge insertion

Remove all edges (x_i, x_j) from E and add $(r(x_i), x_j)$ to the forest.

- Could be done with a `scatter`...
- **But** what happens if we try to insert two edges with the same source?

The trick

- Normalise all edges (x_i, x_j) such that $x_i < x_j$.
- Use `reduce_by_index` to insert edges, picking the highest-indexed target in case of conflict (use `i64.max` as operator).
- Determine which edges (x_i, x_j) were not inserted successfully, update them to $(r(x_i), x_j)$, and try again.
  - Similar to deduplication.

Be careful

Make sure when you insert an edge that the source vertex is currently a root - many ways to do this.

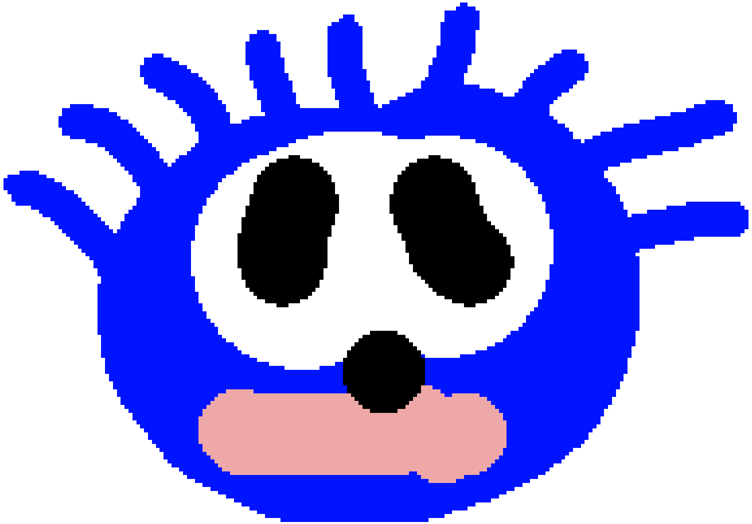
Equivalence classes for region labeling

The problem

Uniquely label connected areas of an image that have the same colour (*regions*).

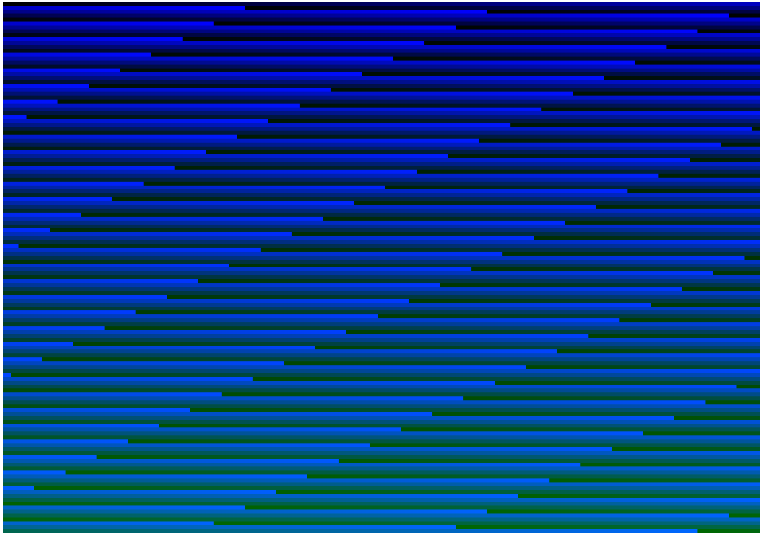
- Widely used in for image analysis and also to implement things like flood filling in image editing programs.
- Can be viewed as an equivalence class partitioning problem — two pixels are in the same equivalence class if they are in the same region.
- We count only *immediate* neighbours (no diagonals).
- An easy solution is *brute force*.

Example of brute force region labeling



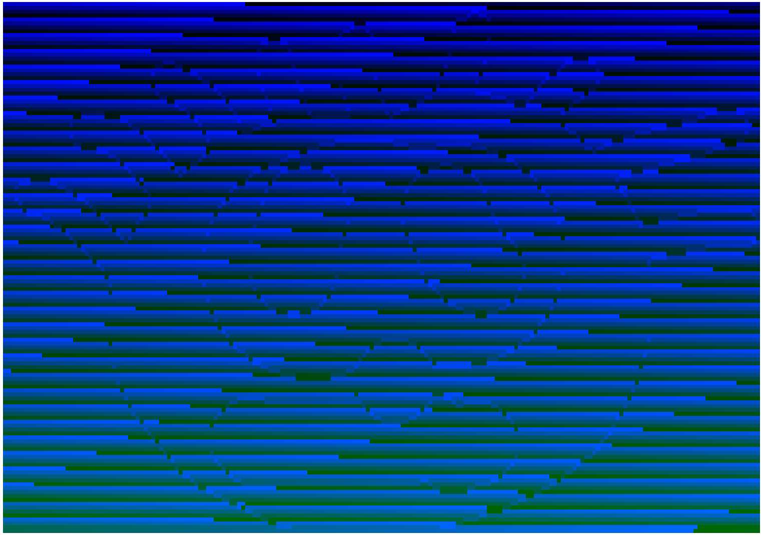
Initial artwork.

Example of brute force region labeling



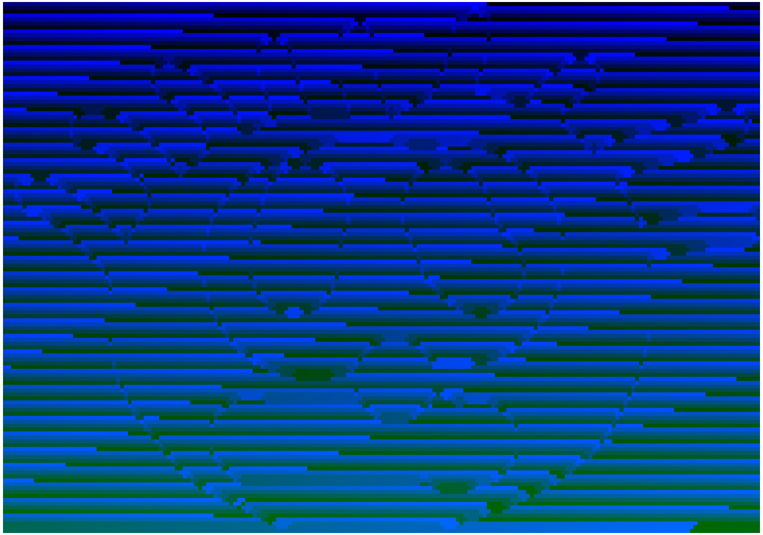
Labeling each pixel (and turning each label into a colour).

Example of brute force region labeling



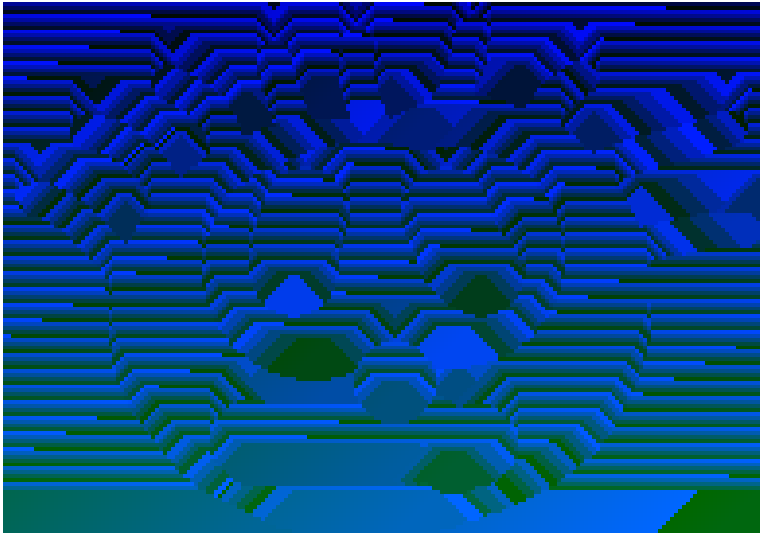
Iteration 1.

Example of brute force region labeling



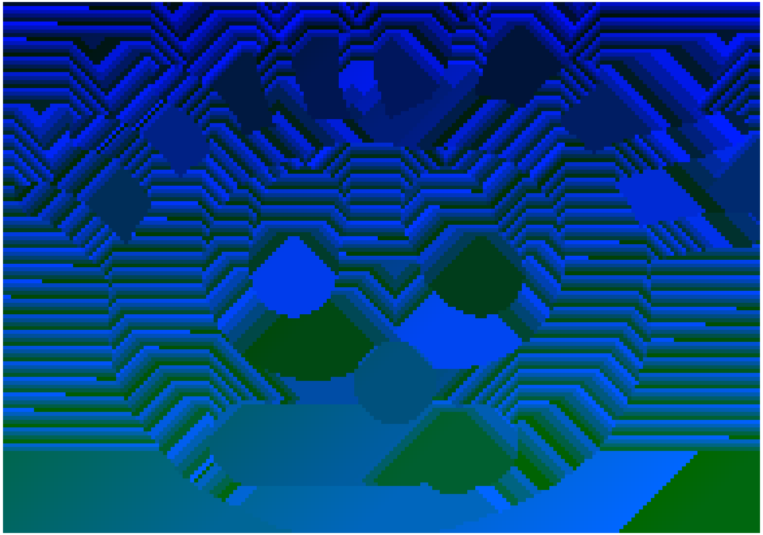
Iteration 2.

Example of brute force region labeling



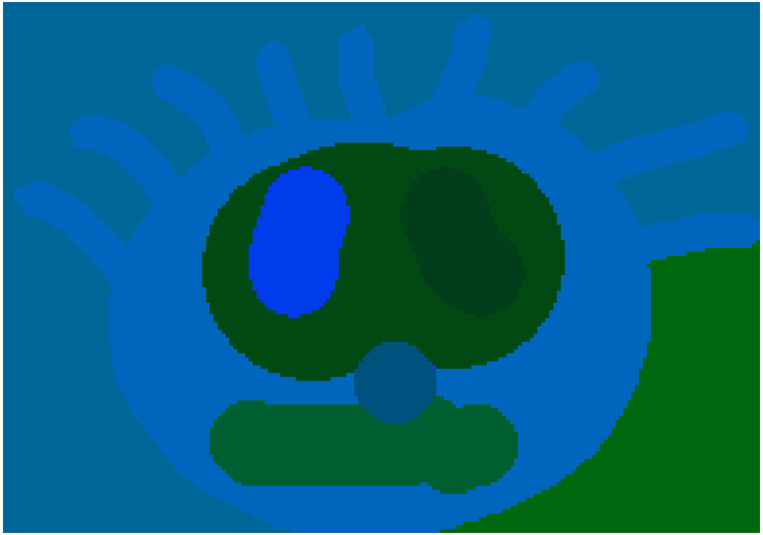
Iteration 10.

Example of brute force region labeling



Iteration 20.

Example of brute force region labeling



Final result.

Doing better (this is your assignment!)

1. Maintain a forest $P : [h * w]i64$ where $P[i * w + j]$ is the parent of pixel (i, j).
2. A pixel has edges to those of its four neighbours which has the same colour.
3. Insert all the edges using the parallel algorithm.
4. Look up the representative element of each pixel by following path to the root.

Things to watch out for

- Make sure that the edges are normalised (go from lower index to higher index).
- Convert two-dimensional pixel indexes to one dimensional, so `reduce_by_index` will work.
- Take care not to overwrite non-roots!

There are multiple solutions to these problems, with different tradeoffs.