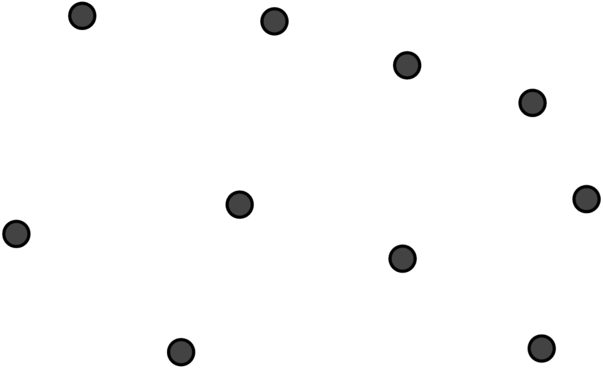
Annual progress review
Fixing Data Parallel Haskell's space complexity

Amos Robinson

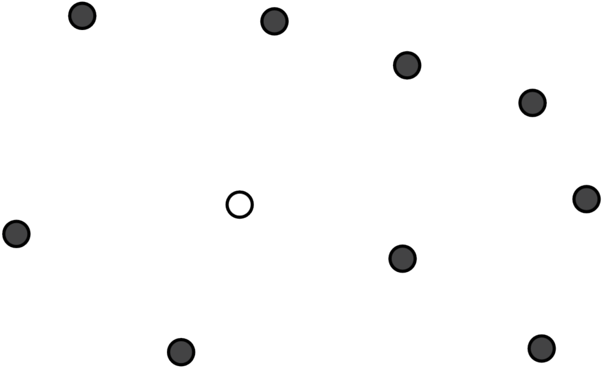
March 12, 2014

Closest points



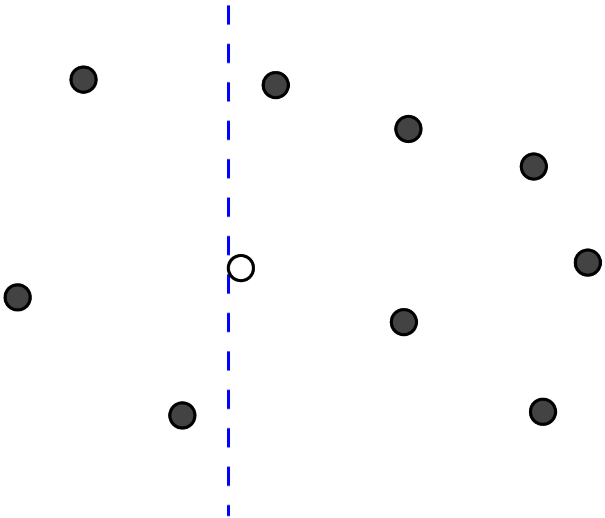
Given an array of points, how do we find the two closest ones?

Closest points



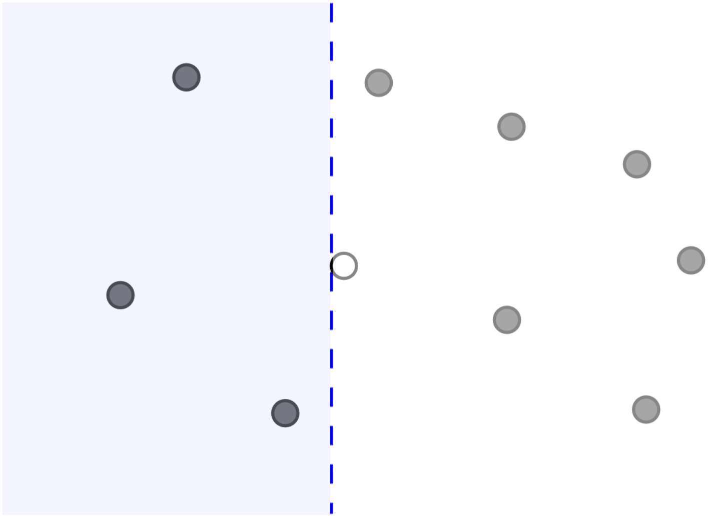
Choose a random point

Closest points



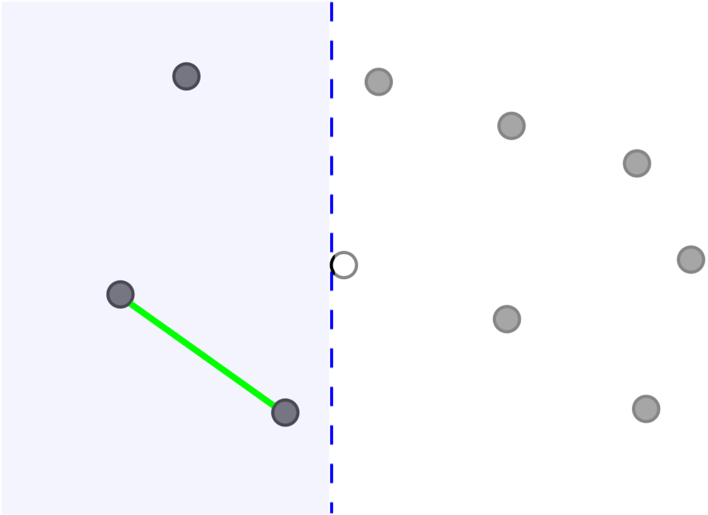
Split into two groups - left of point, right of point

Closest points



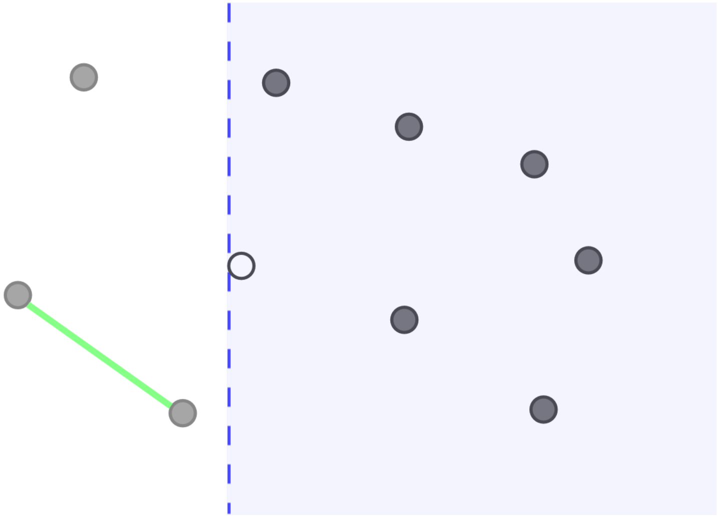
Let's look at the left group first

Closest points



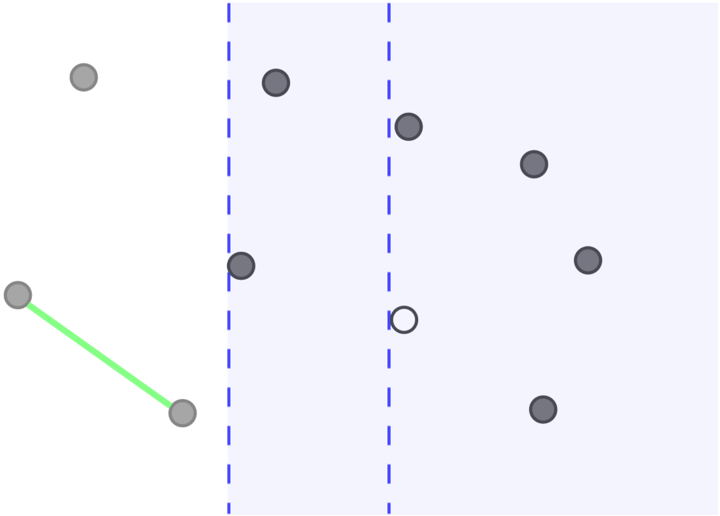
There are only 3 points, so just look at them all (n^2)

Closest points



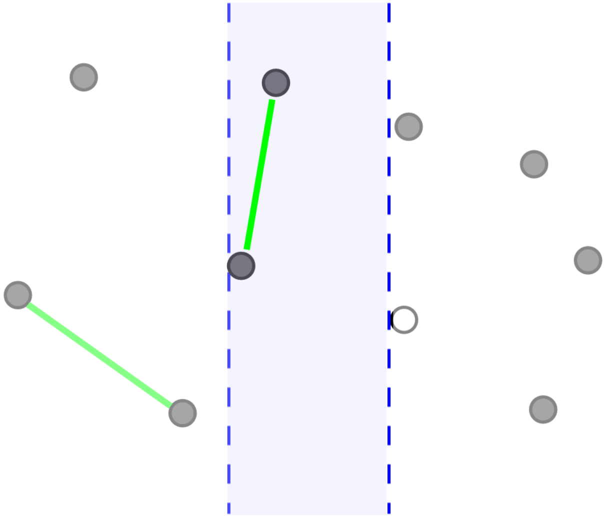
Look at the right set now. There are 7, including the split point.

Closest points



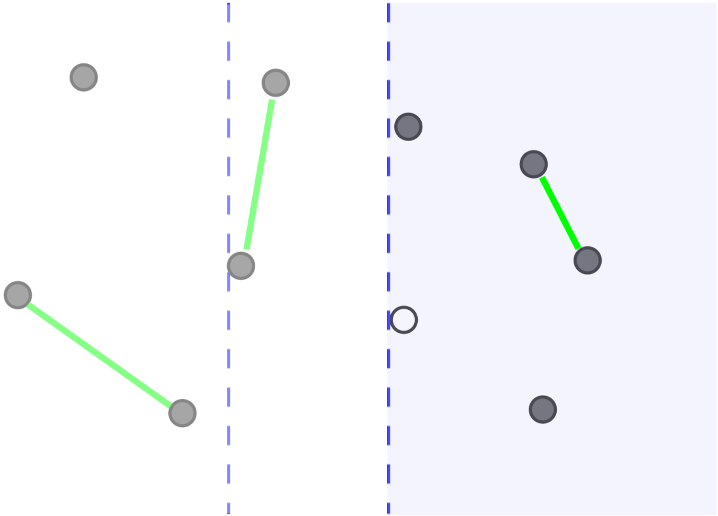
Choose another random point and perform another split

Closest points



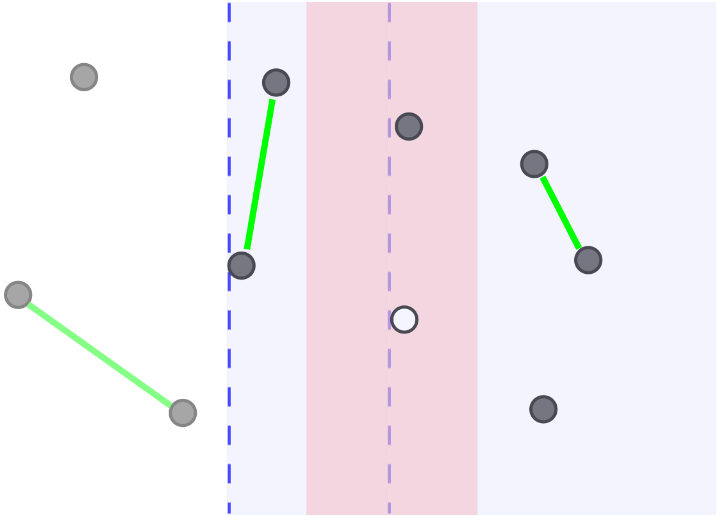
Only 2, so do n^2

Closest points



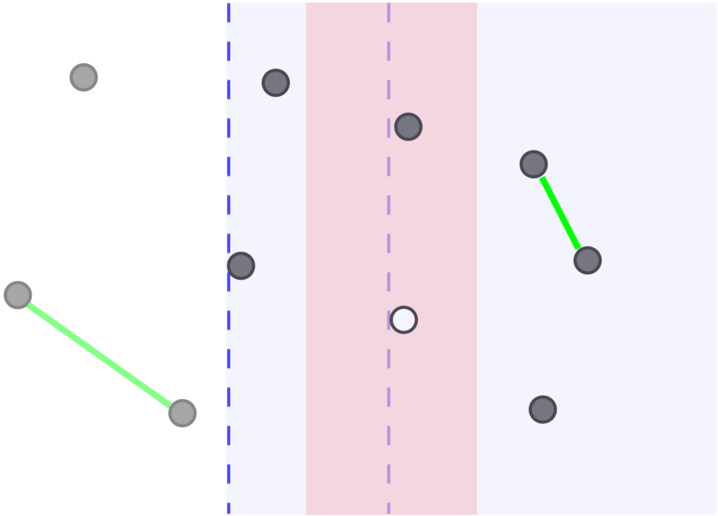
Only 5, so do n^2

Closest points



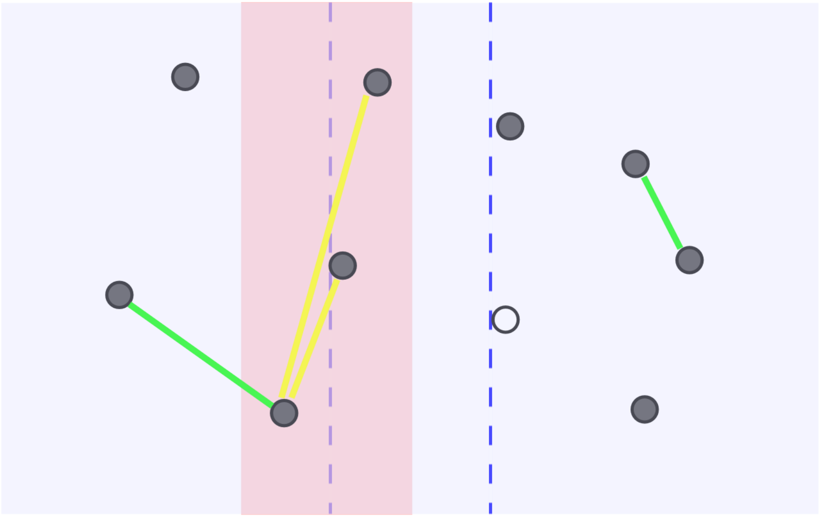
We're merging two sets, but we only need to look at the borders

Closest points



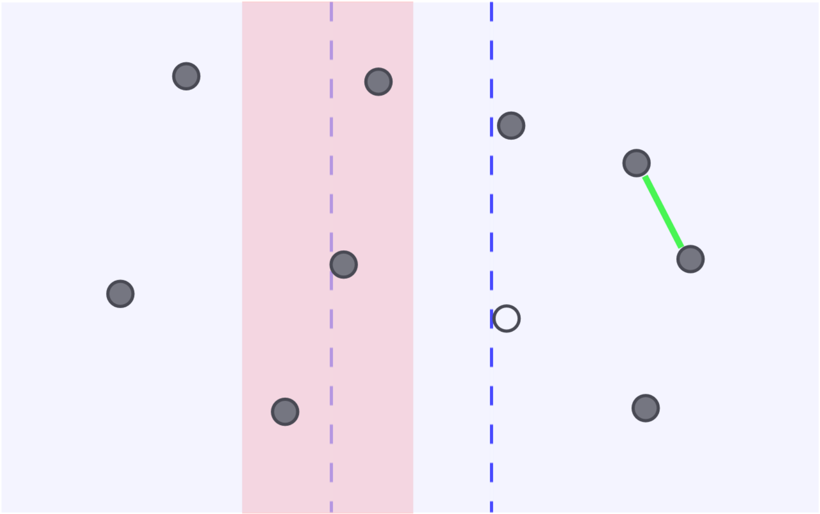
There are no points near the border on the left

Closest points



Check points near the border

Closest points



And we have found the two closest points

How can we parallelise this?

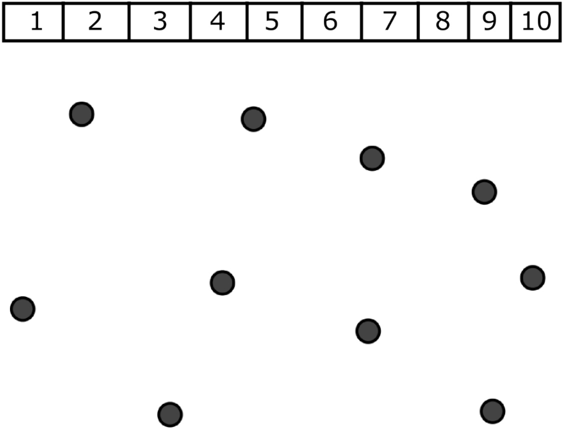
Simple - fork off a new thread for each branch

- Fork a certain number under a threshold - *unbalanced!*
- Dynamic scheduling - *runtime communications overheads!*

Nested data parallelism

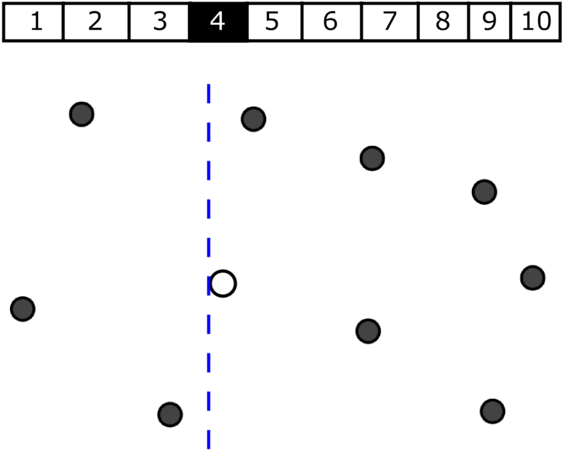
Nested data parallelism lets us do this in a balanced way, by collecting computations into larger chunks, and running in parallel.

Nested data parallelism



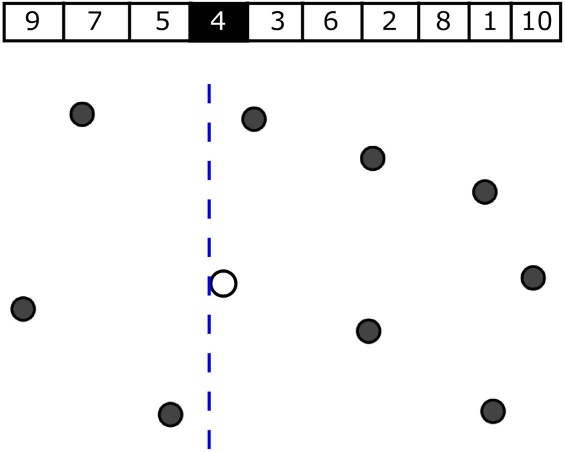
Given the array of points

Nested data parallelism



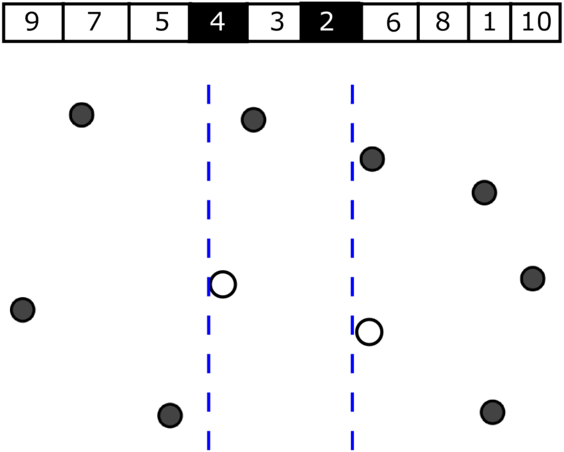
Select a random point

Nested data parallelism



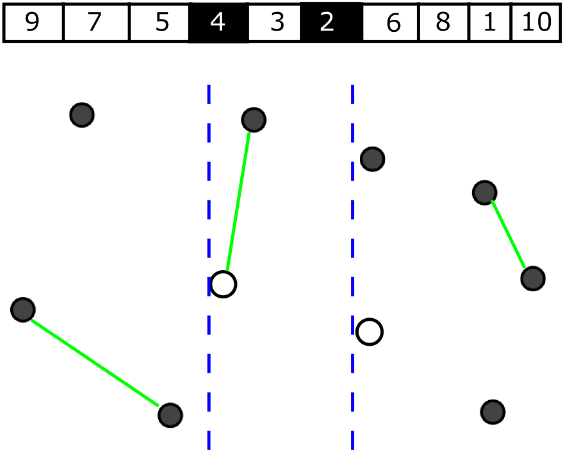
Reorder - put those to the left before those to the right (parallel)

Nested data parallelism



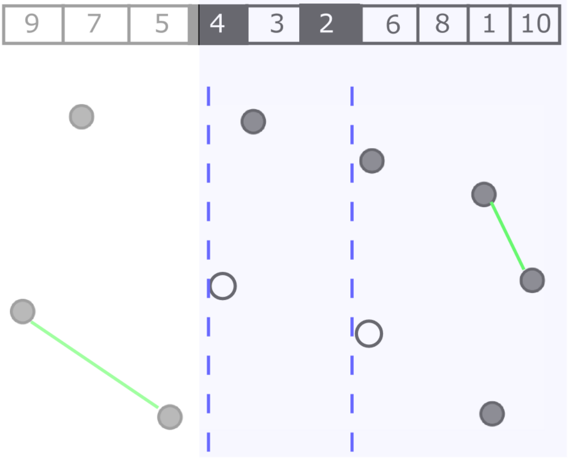
Another random point and reorder (parallel)

Nested data parallelism



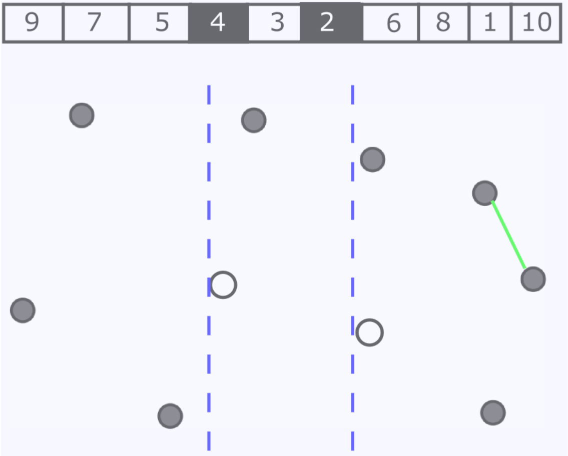
Perform small ones in parallel

Nested data parallelism



Merge upwards (parallel)

Nested data parallelism

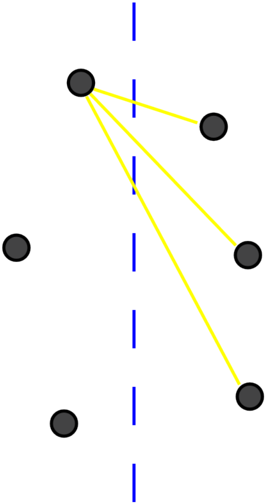


Merge upwards (parallel)

Sequential merging

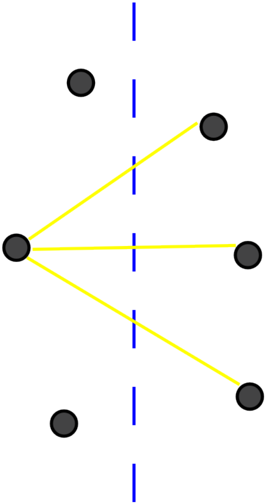
When we are merging some boundary points sequentially, we allocate several small arrays in sequence

Sequential merging



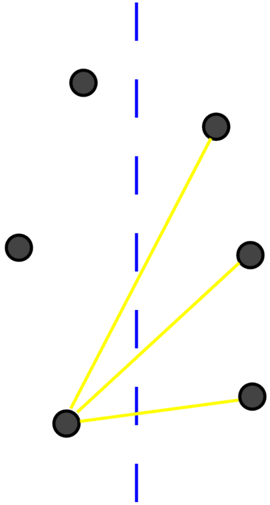
```
[ dist 1 4, dist 1 5, dist 1 6 ]
```

Sequential merging



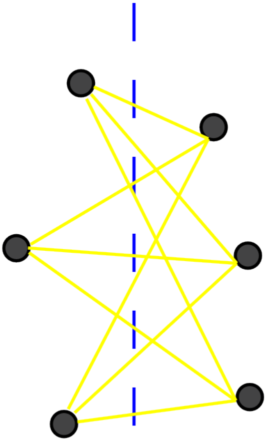
```
[ dist 2 4, dist 2 5, dist 2 6 ]
```

Sequential merging



```
[ dist 3 4, dist 3 5, dist 3 6 ]
```

Parallel merging



However, to perform it in parallel, we allocate *all* arrays at once

```
[ [dist 1 4, dist 1 5, dist 1 6]
, [dist 2 4, dist 2 5, dist 2 6]
, [dist 3 4, dist 3 5, dist 3 6] ]
```

How do we fix this?

Array fusion can remove arrays!

DPH already has fusion

Stream fusion:

- But it relies on compiler heuristics, and is not predictable
- Nor is it optimal

Stream fusion

For this example, stream fusion requires three loops instead of one

```
filterMax (vs : Vector Int) =
 let vs' = map     (+1)     vs
     m   = fold       0 max vs'
     vs''= filter (>0)     vs'
 in (m, vs'')
```

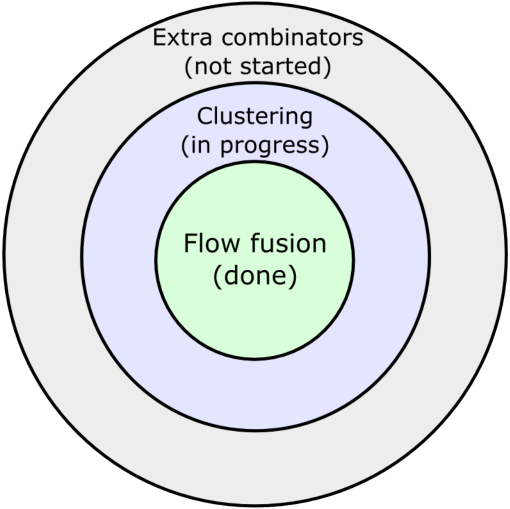
Stream fusion is insufficient

We want a more ‘principled’ approach, with strong guarantees about what will be fused

My work

My Work

My work - outline



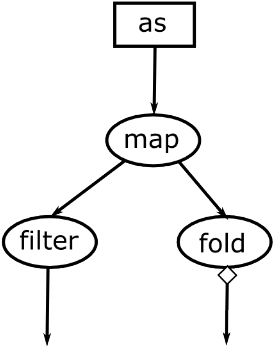
Flow fusion

So, we implemented ‘flow fusion’

- Combinator-based
- Given a set of combinators, *if* they can be fused into a single loop, they *will*
- And we wrote a paper for Haskell Symposium ’13: Data flow fusion with series expressions in Haskell

Flow fusion - example

```
filterMax (vs : Vector Int) =
 let vs' = map     (+1)     vs
     m   = fold      0 max vs'
     vs''= filter (>0)     vs'
 in (m, vs'')
```



This becomes a single imperative loop.

Clustering

Given a set of combinators, partition into as few clusters as possible

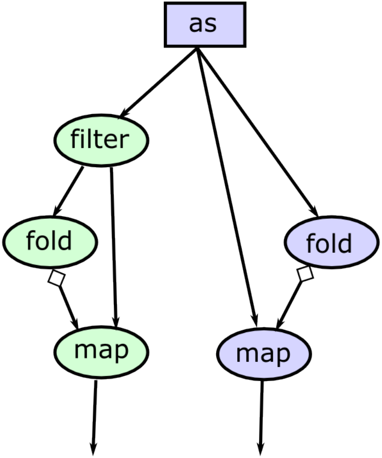
- This is standard fusion
- Lots of literature on this for imperative programs
- Except we also have interesting combinators like *filter*

Clustering - multiple solutions

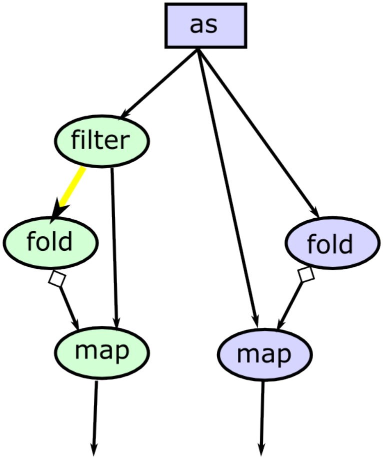
- There are *many* possible clusterings for a given graph
- We want to find the *best*
- NP-hard, but the problems are relatively small
- Integer linear programming!

Clustering - example

```
normalise2 (as : Vector Int) =
 let filt  = filter (>0)   as
     s1    = fold   (+) 0 filt
     s2    = fold   (+) 0 as
     filt' = map    (/s1) filt
     as'   = map    (/s2) as
 in (filt', as')
```

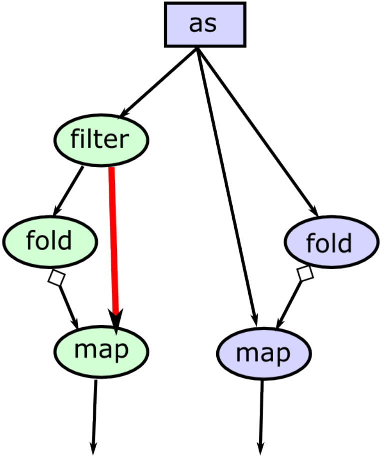


Clustering - example



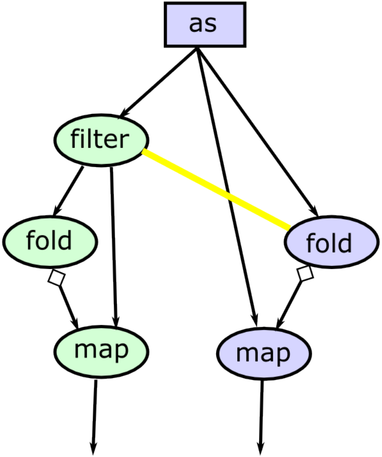
We can fuse these.

Clustering - example



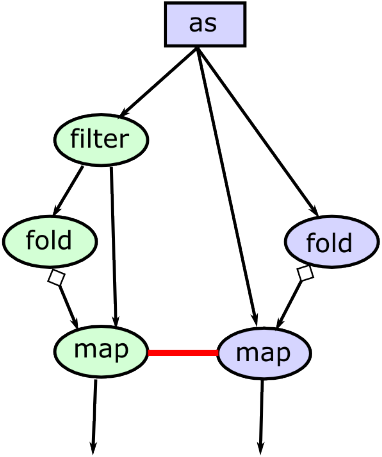
No fusion between *filter* and *map*, because the *fold* is in the way.

Clustering - example



We can fuse these, despite being different colours.

Clustering - example



We *can't* fuse these, as they have different sized inputs

Clustering - current status

- Have implemented prototype ILP formulation
- Some optimisations to algorithm are possible
- Prove correctness
- Implement real version and integrate into GHC

Extra combinators

- Current version only supports certain combinators
  map, *filter*, *fold*, *gather*
- Data Parallel Haskell requires many more
  generate, *concat*, *scan*, *segmented map*, *segmented filter*,...
- But I know the general idea now

Summary - the problem

Nested data parallelism can expose too much parallelism, leading to space complexity problems.
Fusing loops removes the large arrays, but the current fusion system is sub-optimal and unpredictable.
We need a fusion system with strong guarantees about which arrays will be fused away.

Summary - progress

What have I done this year?

- Fixed memory and termination problems with SpecConstr optimisation in GHC
- Flow fusion Haskell Symposium paper
- Implemented ILP formulation for clustering