Parallel Functional Programming
More on the Par Monad
Lecture 5

Mary Sheeran

(with thanks to Simon Marlow for reuse of slides, the blue ones, and of code)

http://www.cse.chalmers.se/edu/course/pfp

Note

- There is now a Google group for the course. Please join. News will now appear there.
- The first lab, part one is up. It is time to get working! Groups of 2 are the norm.
- I need a Chalmers student to be a class rep.
  - (e.g. one who fancies working for Klarna writing Erlang programs rather than doing a doctorate)

In the beginning were

```
par :: a -> b -> b
pseq :: a -> b -> b
```

- pseq expresses sequential evaluation order

\+ par turns a lazy computation into a future

\- par demands operational understanding of execution
(see rules on next slides)

Rules for par (from Par Monad paper)

You must

(a) pass an unevaluated computation to par

(b) ensure that its value will not be required by the enclosing computation for a while, and

(c) ensure that the result is shared by the rest of the program.

reasoning about par

- there is an op. semantics of par in [Baker-Finch et al, 2000]
  but it is for Core, and the compiler munges a program a lot before it gets to core

(Aside : there is clearly plenty of research needed here
Dave Sand's improvement theory could provide inspiration,
any takers? hard!)

Laziness and the need to reason about it may reduce usability of par

Evaluation strategies

The Eval monad allows programmer to express ordering of par and pseq (an improvement over using raw form)

Evaluation strategies provide another layer of abstraction and help avoid some (but not all pitfalls)

User of strategies need to write functions that consume lazy data structures, so problems remain, particularly for larger examples

Enter the Par Monad

From the Haskell'11 paper:

> Our goal with this work is to find a parallel programming model that is expressive enough to subsume Strategies, robust enough to reliably express parallelism, and accessible enough that non-expert programmers can achieve parallelism with little effort

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a -> a

fork :: Par () -> Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

forking is *explicit*

results are communicated through IVars

slide by Simon Marlow

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a -> a

fork :: Par () ->

data IVar
new :: Par (IVar
get :: IVar a ->
put :: NFData a => IVar a -> a -> Par ()
```

Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

semantics of fork:

the argument computation (child) is executed concurrently with the current computation (the parent)

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a ->

fork :: Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

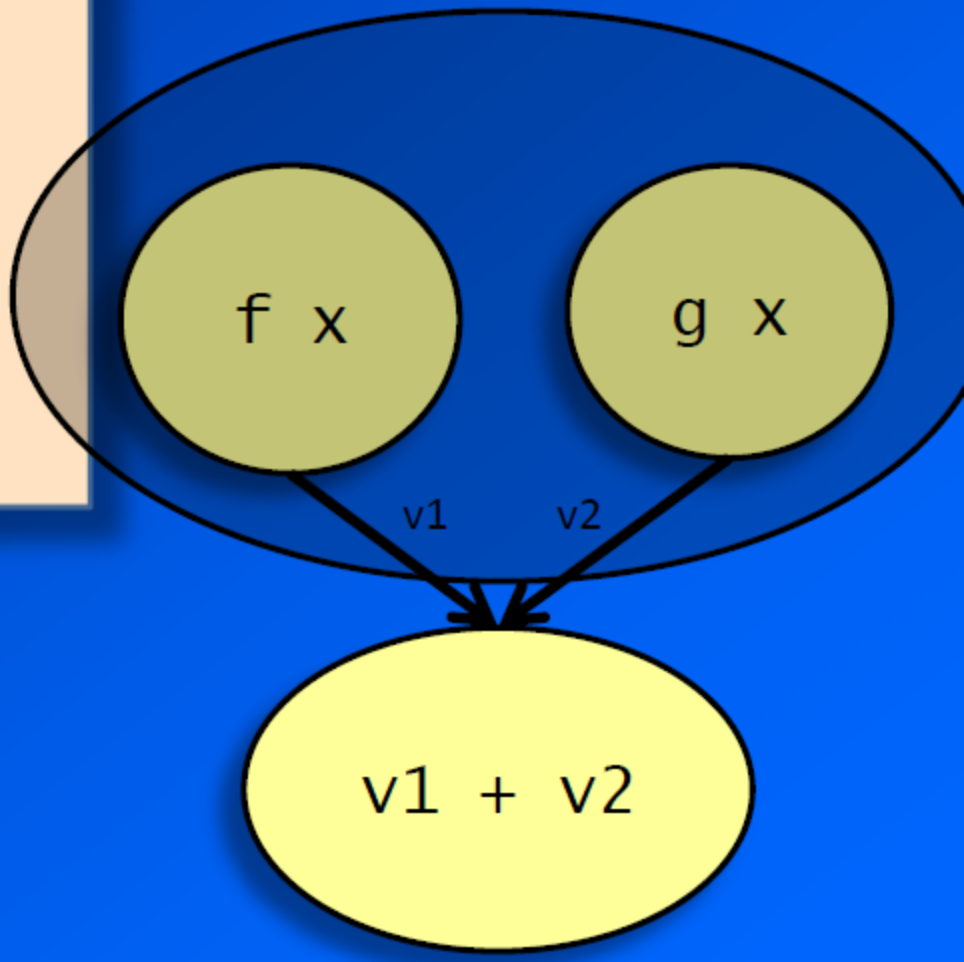
Par is a monad for parallel computation

this is how results are communicated from the child back to the parent

results are communicated through IVars

A bit more complex...

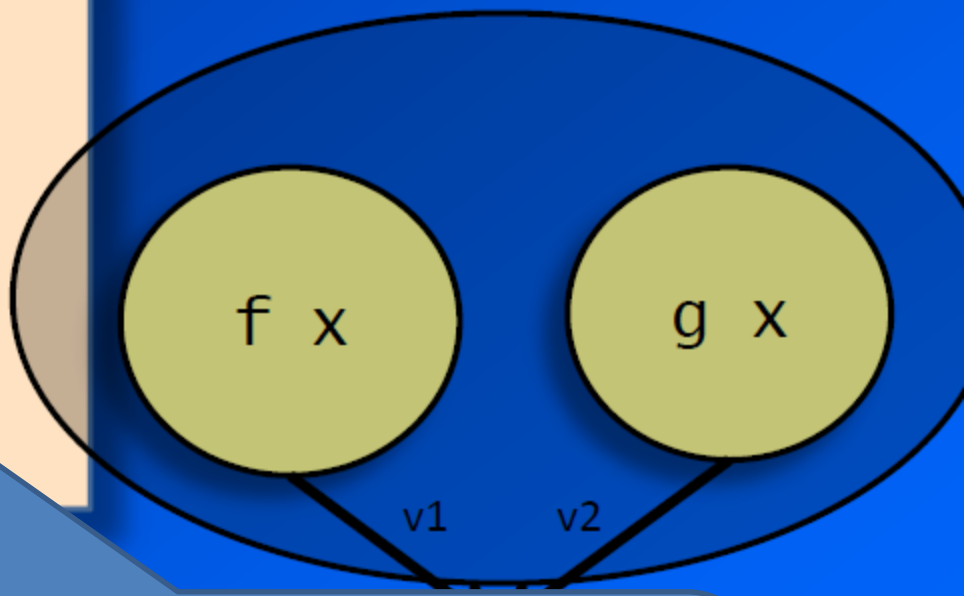
```
do v1 <- new
   v2 <- new
   fork $ put v1 (f x)
   fork $ put v2 (g x)
   get v1
   get v2
   return (v1 + v2)
```



slide by Simon Marlow

A bit more complex...

```
do v1 <- new
   v2 <- new
   fork $ put v1 (f x)
   fork $ put v2 (g x)
   get v1
   get v2
   return (v1 + v2)
```

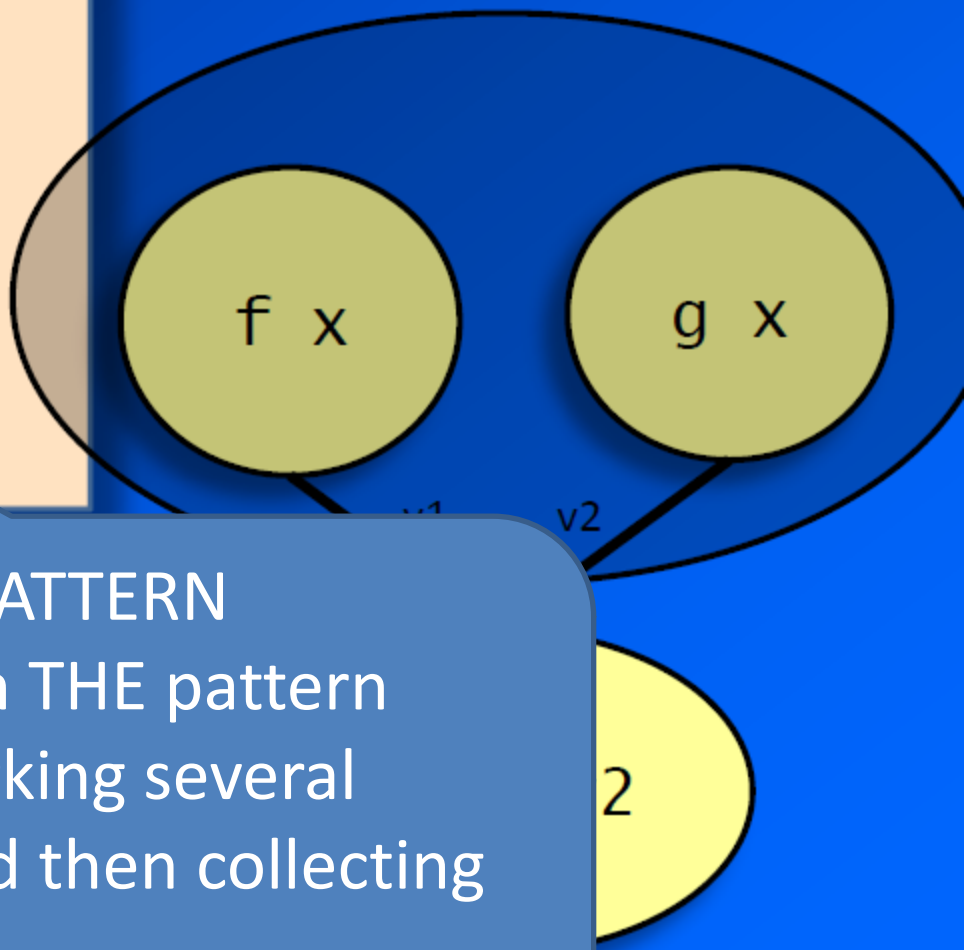


Note that put is fully strict
(=> normal form data NFData context)

Stuff flowing along arcs is fully evaluated

A bit more complex...

```
do v1 <- new
   v2 <- new
   fork $ put v1 (f x)
   fork $ put v2 (g x)
   get v1
   get v2
   return (v1 +
```



A PATTERN

maybe even THE pattern

a parent forking several children and then collecting results

Capture it

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

Capture it

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

First one child

The Ivar represents a computation that will complete later (a future)

Capture it

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

spawn subsumes fork,new,put

prevents errors involving too many puts (runtime errors)

still sometimes want to use fork etc. (see type inference ex.)

Capture it

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

Capture it

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

common pattern: spawn a process for each element of the input list to apply f to that input. Wait for results.

saw parMap with the f having type (a-> b) last time

Capture it

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

Version in library works for any Traversble data structure, not just lists

Dataflow problems

- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

slide by Simon Marlow

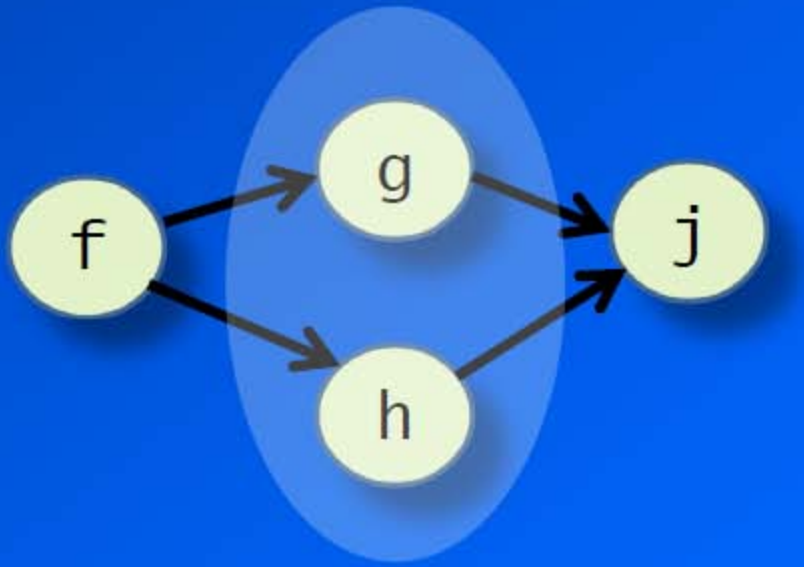
Example

- Consider typechecking (or inferring types for) a set of non-recursive bindings.
- Each binding is of the form `x = e` for variable x, expression e
- To typecheck a binding:
  - input: the types of the identifiers mentioned in e
  - output: the type of x
- So this is a dataflow graph
  - a node represents the typechecking of a binding
  - the types of identifiers flow down the edges
  - It's a *dynamic* dataflow graph: we don't know the shape beforehand

slide by Simon Marlow

Example

```
f = ...
g = ... f ...
h = ... f ...
j = ... g ... h ...
```



slide by Simon Marlow

Implementation

- We parallelised an existing type checker (nofib/infer).
- Algorithm works on a single term:

```
data Term = Let VarId Term Term | ...
```

- So we parallelise checking of the top-level Let bindings.

```
let x1 = e1 in
let x2 = e2 in
let x3 = e3 in
...
```

slide by Simon Marlow

The parallel type inferencer

- Given:

```
inferTopRhs :: Env -> Term -> PolyType
makeEnv :: [(VarId,Type)] -> Env
```

- We need a type environment:

```
type TopEnv = Map VarId (IVar PolyType)
```

- The top-level inferencer has the following type:

```
inferTop :: TopEnv -> Term -> Par MonoType
```

slide by Simon Marlow

Parallel type inference

```
inferTop :: TopEnv -> Term -> Par MonoType
inferTop topenv (Let x u v) = do
    vu <- new

    fork $ do
      let fu = Set.toList (freeVars u)
      tfu <- mapM (get . fromJust . flip Map.lookup topenv) fu
      let aa = makeEnv (zip fu tfu)
      put vu (inferTopRhs aa u)

    inferTop (Map.insert x vu topenv) v


inferTop topenv t = do
  -- the boring case: invoke the normal sequential
  -- type inference engine
```

slide by Simon Marlow

Parallel type inference

```
inferTop :: TopEnv -> Term -> Par MonoType
inferTop topenv (Let x u v) = do
    vu <- new

    fork $ do
      let fu = Set.toList (freeVars
      tfu <- mapM (get . fromJust
      let aa = makeEnv (zip fu t
      put vu (inferTopRhs aa

    inferTop (Map.insert x vu top


inferTop topenv t = do
  -- the boring case: invoke the
  -- type inference engine
```

Note:
involves explicit fork, get ,put

slide by Simon Marlow

Results

```
let id = \x.x in
    let x = \f.f id id in
    let x = \f . f x x in
    let x = \f . f x x in
    let x = \f . f x x in
    ...
    let x = let f = x in \z . z in
    let y = \f.f id id in
    let y = \f . f y y in
    let y = \f . f y y in
    let y = \f . f y y in
    ...
    let x = let f = y in \z . z in
    \f. let g = \a. a x y in f
```

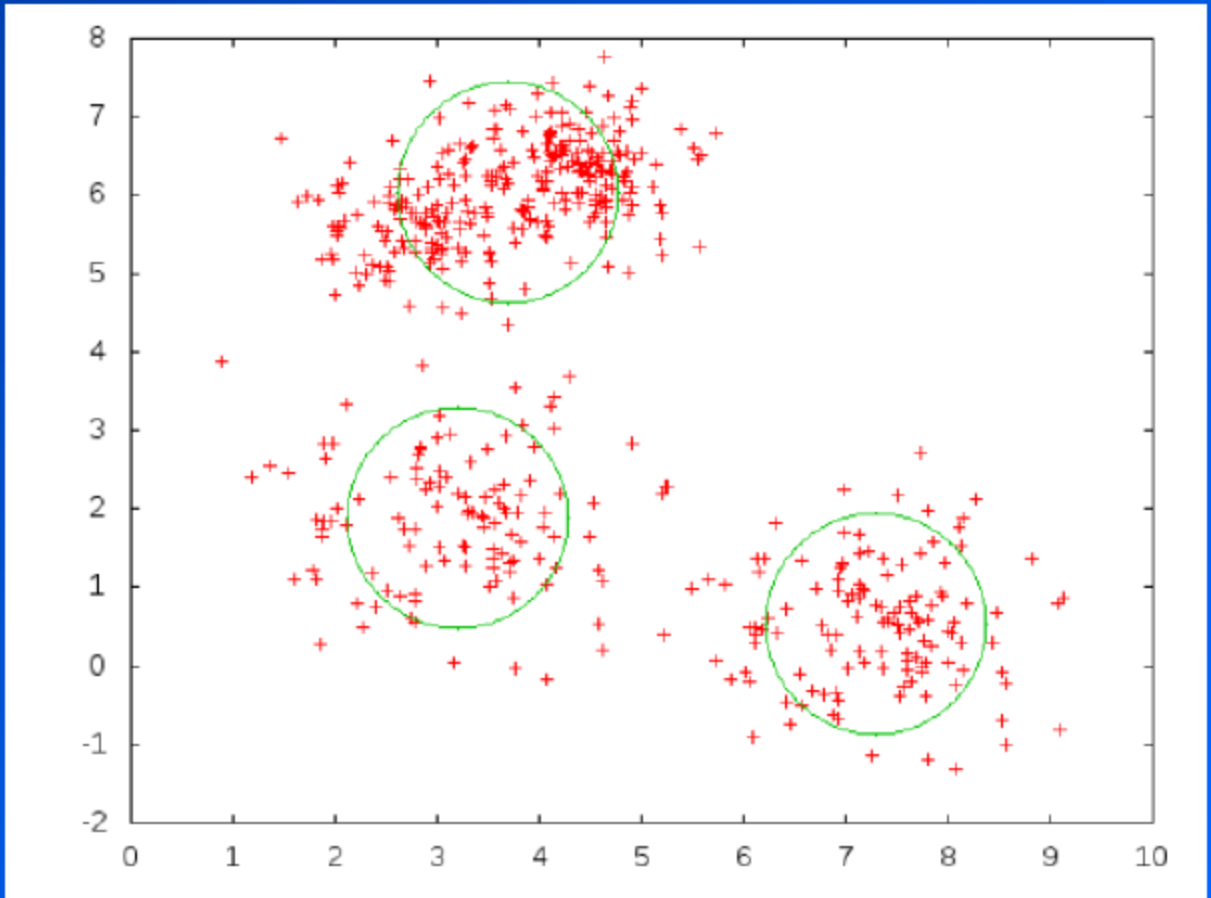
- -N1: 1.12s
- -N2: 0.60s (1.87x speedup)
- available parallelism depends on the input: these bindings only have two branches

slide by Simon Marlow

Exercise: K-Means

slide by Simon Marlow

- A data-mining algorithm, to identify clusters in a data set.



K-Means

- We use a heuristic technique (Lloyd's algorithm), based on iterative refinement.

  1. Input: an initial guess at each cluster location
  2. Assign each data point to the cluster to which it is closest
  3. Find the *centroid* of each cluster (the average of all points)
  4. repeat 2-3 until clusters stabilise

- Making the initial guess:
  1. Input: number of clusters to find
  2. Assign each data point to a random cluster
  3. Find the centroid of each cluster

- Careful: sometimes a cluster ends up with no points!

slide by Simon Marlow

kmeans code

```
data Vector = Vector Double Double Double
deriving (Show,Read,Typeable,Data,Eq)
```

kmeans code

```
data Vector = Vector Double Double Double
deriving (Show,Read,Typeable,Data,Eq)
```

Actually there are {-#UNPACK#-} ! annotations before the Doubles

kmeans code

```
data Cluster = Cluster
               {
                  clId    :: {-#UNPACK#-}!Int,
                  clCount :: {-#UNPACK#-}!Int,
                  clSum   :: {-#UNPACK#-}!Vector,
                  clCent  :: {-#UNPACK#-}!Vector
               }  deriving
(Show,Read,Typeable,Data,Eq)

instance NFData Cluster  -- default should be fine
```

kmeans code

```
sqDistance :: Vector -> Vector -> Double
sqDistance (Vector x1 y1 z1) (Vector x2 y2 z2)
  = ((x1-x2)^2) + ((y1-y2)^2 + (z1-z2)^2)
```

```
makeCluster :: Int -> [Vector] -> Cluster
makeCluster clid vecs
   = Cluster { clId = clid,
               clCount = count,
               clSum = vecsum,
               clCent = centre
             }
   where vecsum@(Vector a b c)  = foldl' addVector zeroVector vecs
         centre = Vector (a / fromIntegral count)
                         (b / fromIntegral count)
                         (c / fromIntegral count)
         count = length vecs
```

```
-- assign each vector to the nearest cluster centre
assign :: Int -> [Cluster] -> [Vector] -> Array Int [Vector]
assign nclusters clusters points =
    accumArray (flip (:)) [] (0, nclusters-1)
       [ (clId (nearest p), p) | p <- points ]
  where
    nearest p = fst $ minimumBy (compare `on` snd)
                  [ (c, sqDistance (clCent c) p) | c <- clusters ]
```

```
-- assign each vector to the nearest cluster centre
assign :: Int -> [Cluster] -> [Vector] -> Array Int [Vector]
assign nclusters clusters points =
    accumArray (flip (:)) [] (0, nclusters-1)
      [ (clId (nearest p), p) | p <- points ]
  where
    near          $ minimumBy (compare `on` snd)
                    Distance (clCent c) p) | c <- clusters ]
```

```
accumArray
  :: Ix i => (e -> a -> e) -> e -> (i, i) -> [(i, a)] -> Array i e
```

builds an array from list of associations
uses combining function to deal with multiple occurrences of an
an index

```
makeNewClusters :: Array Int [Vector] -> [Cluster]
makeNewClusters arr =
  filter ((>0) . clCount) $
     [ makeCluster i ps | (i,ps) <- assocs arr ]
      -- v. important: filter out any clusters that have
      -- no points.  This can happen when a cluster is not
      -- close to any points.  If we leave these in, then
      -- the NaNs mess up all the future calculations.
```

```
-- Perform one step of the K-Means algorithm

step :: Int -> [Cluster] -> [Vector] -> [Cluster]
step nclusters clusters points
   = makeNewClusters (assign nclusters clusters points)
```

now loop

```
-- K-Means: repeatedly step until convergence

kmeans_seq :: Int -> [Vector] -> [Cluster] -> IO
[Cluster]
kmeans_seq nclusters points clusters = do
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
         = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let clusters' = step nclusters clusters points
        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters'
  --
  loop 0 clusters
```

How to parallelise?

`assign` ? since it is just a map over points?

doesn't get us far

cannot parallelise `accumArray` directly

would need to do multiple `accumArrays`

```
-- assign each vector to the nearest cluster centre
assign :: Int -> [Cluster] -> [Vector] -> Array Int [Vector]
assign nclusters clusters points =
    accumArray (flip (:)) [] (0, nclusters-1)
       [ (clId (nearest p), p) | p <- points ]
  where
    nearest p = fst $ minimumBy (compare `on` snd)
                  [ (c, sqDistance (clCent c) p) | c <- clusters ]
```

How to parallelise?

`makeNewClusters` ? easy because each `makeNewCluster` is independent of the others

doesn't get us far

not many clusters => not much parallelism

```
makeNewClusters :: Array Int [Vector] -> [Cluster]
makeNewClusters arr =
  filter ((>0) . clCount) $
     [ makeCluster i ps | (i,ps) <- assocs arr ]
      -- v. important: filter out any clusters that have
      -- no points.  This can happen when a cluster is not
      -- close to any points.  If we leave these in, then
      -- the NaNs mess up all the future calculations.
```

think at a higher level

from Marlow's CEFP notes

We would like a way to parallelise the problem at a higher level. That is, we would like to divide the set of points into chunks, and process each chunk in parallel, somehow combining the results. In order to do this, we need a combine function, such that

```
points == as ++ bs
==>
step n cs points == step n cs as `combine` step n cs bs
```

combining two clusters

We are summing vectors and counting data points, so everything works
(the magic of associative, commutative operators)

```
combineClusters c1 c2 =
  Cluster {clId = clId c1,
           clCount = count,
           clSum = vecsum,
           clCent = Vector (a / fromIntegral count)
                           (b / fromIntegral count)
                           (c / fromIntegral count)}
  where count = clCount c1 + clCount c2
        vecsum@(Vector a b c)  = addVector (clSum c1) (clSum c2)
```

```
reduce :: Int -> [[Cluster]] -> [Cluster]
reduce nclusters css =
  concatMap combine $ elems $
     accumArray (flip (:)) [] (0,nclusters)
                [ (clId c, c) | c <- concat css]
 where
  combine [] = []
  combine (c:cs) = [foldr combineClusters c cs]
```

processing N chunks of the data space independently, and each returns a set of clusters

Need to reduce N sets of sets of clusters to a single set (another accumArray)

Done!

Now can use parMap to invoke step on each chunk

followed by reduce to combine the results

```
-- K-Means: repeatedly step until convergence (Par monad)

kmeans_par :: Int -> Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_par mappers nclusters points clusters = do
  let chunks = split mappers points
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
        = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let
            new_clusterss = runPar $ Par.parMap (step nclusters clusters) chunks

            clusters' = reduce nclusters new_clusterss

        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters'
  --
  final <- loop 0 clusters
```

reminder of original code

```
-- K-Means: repeatedly step until convergence

kmeans_seq :: Int -> [Vector] -> [Cluster] -> IO
[Cluster]
kmeans_seq nclusters points clusters = do
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
         = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let clusters' = step nclusters clusters points
        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters'
  --
  loop 0 clusters
```

Parallel

```
-- K-Means: repeatedly step until convergence (Par monad)

kmeans_par :: Int -> Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_par mappers nclusters points clusters = do
  let chunks = split mappers points
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
        = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let
            new_clusterss = runPar $ Par.parMap (step nclusters clusters) chunks

            clusters' = reduce nclus

        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters
  --
  final <- loop 0 clusters
```

parMap ….

reduce …

```
-- K-Means: repeatedly step until convergence (Par monad)

kmeans_par :: Int -> Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_par mappers nclusters points clusters = do
  let chunks = split mappers points
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
        = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let
            new_clusterss = runPar $ Par.parMap (step nclusters clusters) chunks

            clusters' = reduce nclus

        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters
  --
  final <- loop 0 clusters
```

relatively small change to program AFTER modifying the algorithm ☺

```
-- K-Means: repeatedly step until convergence (Par monad)

kmeans_par :: Int -> Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_par mappers nclusters points clusters = do
  let chunks = split mappers points
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
        = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let
            new_clusterss = runPar $ Par.parMap (step nclusters clusters) chunks

            clusters' = reduce nclus

        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters
  --
  final <- loop 0 clusters
```

strategy would be
`using` parList rdeepseq

```
-- K-Means: repeatedly step until convergence (Par monad)

kmeans_par :: Int -> Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_par mappers nclusters points clusters = do
  let chunks = split mappers points
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany
        = do printf "giving up."; return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let
            new_clusterss = runPar $ Par.parMap (step nclusters clusters) chunks

            clusters' = reduce nclus

        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters
  --
  final <- loop 0 clusters
```

scales reasonably well up to 6 cores (3.1 on 4)

Challenge
(no Champagne this time)

- Can you parallelise Barnes-Hut (3D)?

  (see wikipedia, the original paper from 1986 is only 3.5 pages long, and it has a bit of Scheme in the middle to explain the algorithm; talk to Mary if you are interested)

Related work (Par Monad, see paper)

- fork / join Habanero Java, Cilk
- sync. data structures pH, concurrent ML

  Manticore supports both CML model and explict futures
- Intel Concurrent Collections (CnC) provide a superset of Par Monad functionality

Student presentatons

- Single Assignment C
- Manticore
- Cloud Haskell (Erlang ideas in Haskell)
- Intel Concurrent Collections for Haskell
- Spiral
- Many more ….

Talk to Mary if you are interested
Good practice no matter where you plan to end up!

Final words on Par

- runPar is more costly than runEval (but still fairly cheap)
- puts its faith in higher-order skeletons as the means to provide modular parallelism
- See Thursday's lecture by Jost Berthold!

Final words on Par

- Parallel structure is well defined
- Less need to reason about laziness (BUT the sharing of lazy computations between threads is not prevented)
- Doesn’t provide the nice modularity (separation of algorithm and coordination) that strategies does
- All speculative parallelism must be eventually evaluated (unlike in strategies) (to preserve determinism)

Final words on Par

- Par Monad scheduler separate from runtime, easily changed
- Perhaps ordinary mortals should use Par, while par is used for automated parallelisation??
- See Lennart Augustsson's Report from the Real World on May 7. He will likely return to the strict vs lazy question (or rather to the question of controlling evaluation)

Open research problems?

- How to do safe nondeterminism
- implement and compare scheduling algorithms
- better raw performance (integrate more deeply with the RTS)
- Cheaper runPar – one global scheduler

slide by Simon Marlow