Parallel & Concurrent Haskell 2: The Par Monad

Simon Marlow

(Microsoft Research, Cambridge, UK)

Recap

Recap

- In part 1 we learned about
  - The Eval monad
    - nice simple primitives for introducing deterministic parallelism
    - minimal control over the evaluation order
  - Strategies
    - Adding parallelism over (lazy) data structures
    - Composability: combine Strategies into larger ones
    - Modularity: (e `using` s) separates the parallelism from the algorithm

But...

But...

- Lazy evaluation is the magic ingredient that bestows *modularity*, and thus forms the basis of Strategies.
  - but it can be tricky to deal with.

But...

- Lazy evaluation is the magic ingredient that bestows *modularity*, and thus forms the basis of Strategies.
  - but it can be tricky to deal with.
- To use Strategies effectively, you need to understand things like
  - evaluation order (because the argument to rpar must be a lazy computation)
  - garbage collection (because the result of rpar must not be discarded)
  - In a sense this is all tricky *by design* because the Haskell language definition places no requirements on evaluation order or memory behaviour. Compilers are free to do what they like.

But...

- Lazy evaluation is the magic ingredient that bestows *modularity*, and thus forms the basis of Strategies.
  - but it can be tricky to deal with.
- To use Strategies effectively, you need to understand things like
  - evaluation order (because the argument to rpar must be a lazy computation)
  - garbage collection (because the result of rpar must not be discarded)
  - In a sense this is all tricky *by design* because the Haskell language definition places no requirements on evaluation order or memory behaviour. Compilers are free to do what they like.
- Diagnosing performance problems can be hard

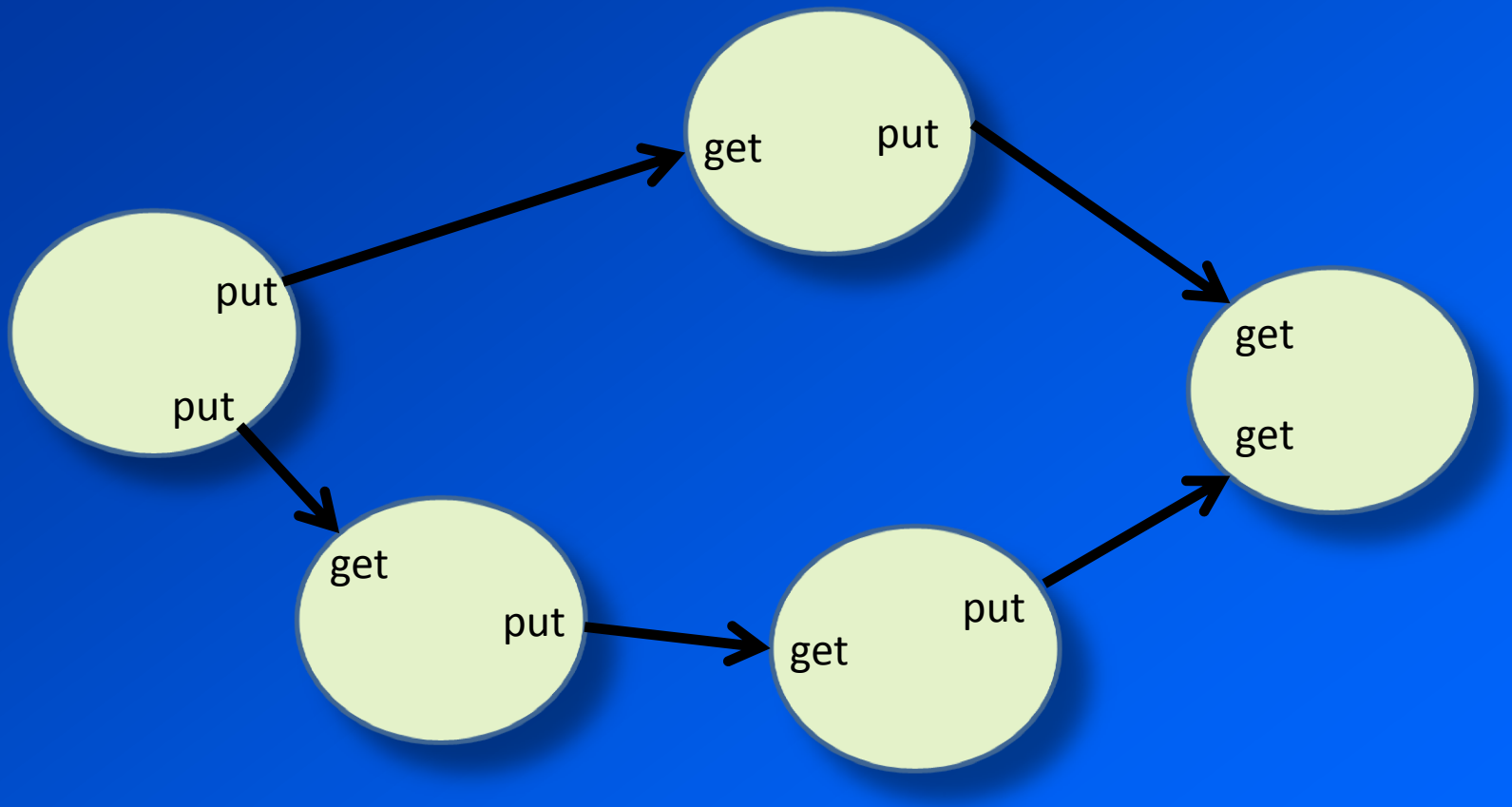
The Par Monad

- Aim for a more direct programming model:
  - sacrifice “modularity via laziness”
  - Avoid the programmer having to think about when things are evaluated
    - … hence avoid many common pitfalls
  - Modularity via *higher-order skeletons*
    - no laziness magic here, just higher-order functions and polymorphism
  - It’s a library written entirely in Haskell
    - Pure API outside, unsafePerformIO + forkIO inside
    - Write your own scheduler!

The basic idea

- Think about your computation as a dataflow graph.

Par expresses dynamic dataflow



The Par Monad

Par is a monad for parallel computation

```
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 ()
```

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)

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*

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

A couple of things to bear in mind

A couple of things to bear in mind

- *put is fully strict*

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

  - all values communicated through IVars are fully evaluated
    - The programmer can tell where the computation is happening, and hence reason about the parallelism
  - (there is a head-strict version put_ but we won't be using it)

A couple of things to bear in mind

- *put is fully strict*

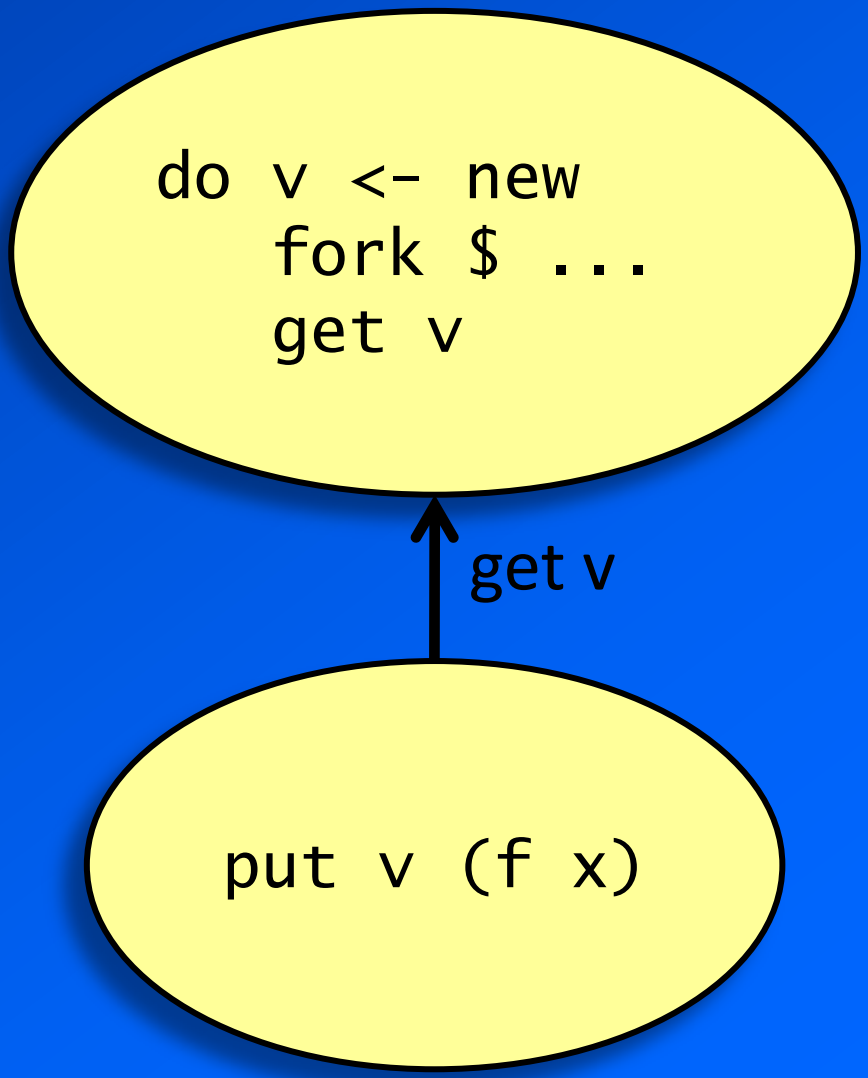
```
put :: NFData a => IVar a -> a -> Par ()
```

  - all values communicated through IVars are fully evaluated
    - The programmer can tell where the computation is happening, and hence reason about the parallelism
  - (there is a head-strict version put_ but we won't be using it)
- *put twice on the same IVar is an error*
  - This is a requirement for Par to be deterministic

How does this make a dataflow graph?

```
do v <- new
   fork $ put v (f x)
   get v
```

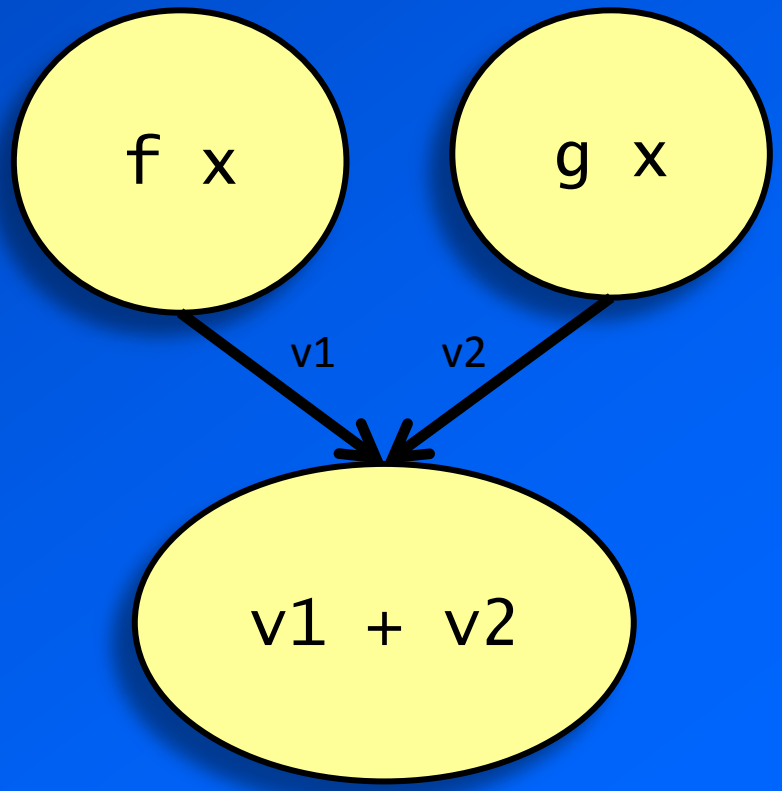
- fork creates a new node in the graph
- get creates a new edge
  - from the node containing the put
  - to the node containing the get



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)
```

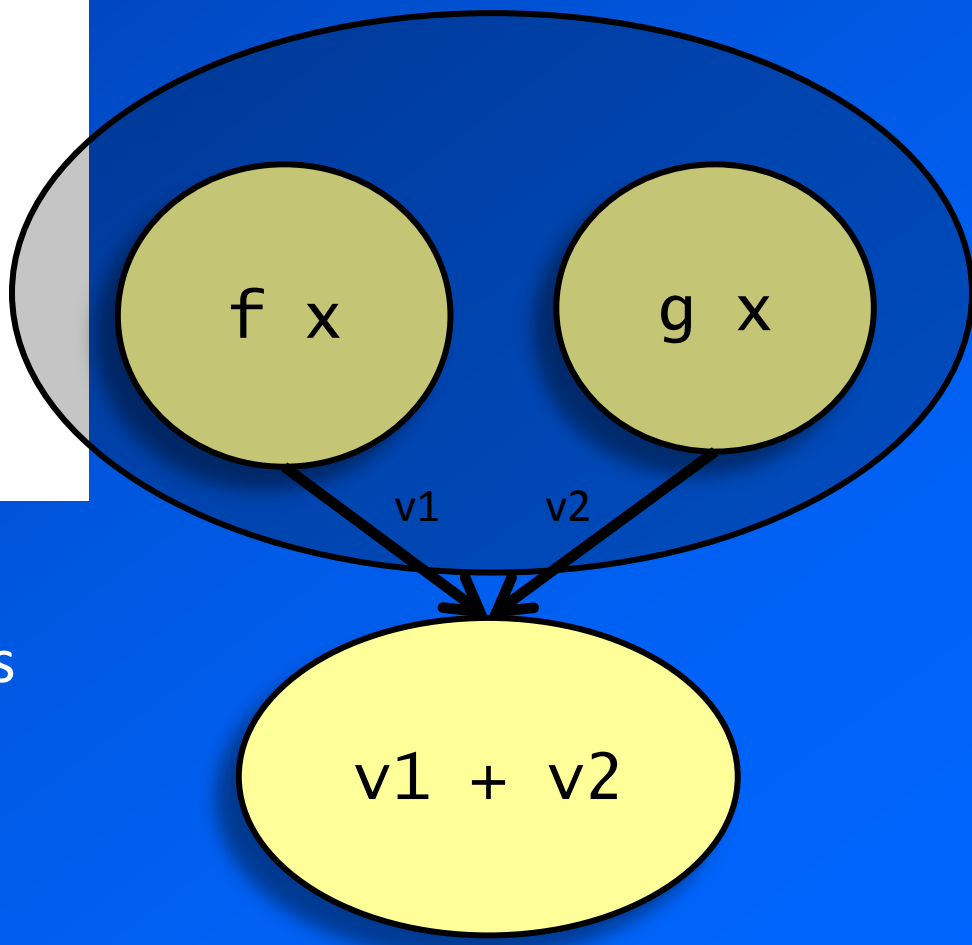
- runPar evaluates the graph
- nodes with no dependencies between them can execute in parallel



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)
```

- runPar evaluates the graph
- nodes with no dependencies between them can execute in parallel



Back to the Sudoku example

- The sequential code:

```
main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    print $ length $ filter isJust $ map solve grids
```

```
solve :: String -> Maybe Grid
```

Sudoku solver, version 2

- Divide the work in two:

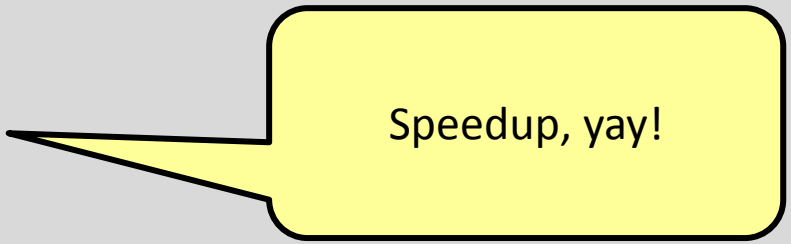
```
import Control.Monad.Par

main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f

    let (as,bs) = splitAt (length grids `div` 2) grids

    print $ length $ filter isJust $ runPar $ do
       i1 <- new
       i2 <- new
       fork $ put i1 (map solve as)
       fork $ put i2 (map solve bs)
       as' <- get i1
       bs' <- get i2
       return (as' ++ bs')
```

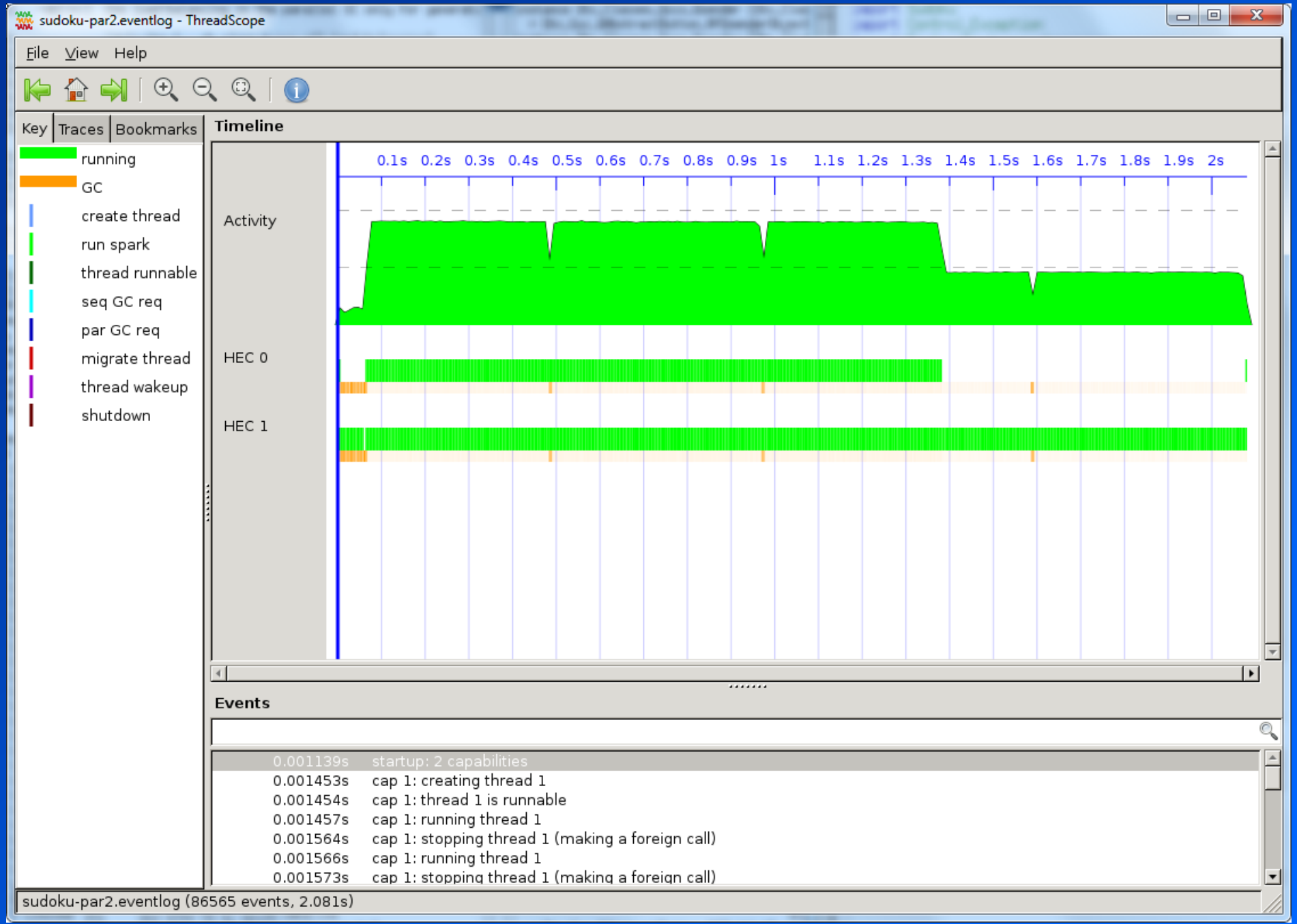
Compile it for parallel execution

```
$ ghc --make -O2 sudoku-par2.hs -rtsopts -threaded
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku-par2.hs, sudoku-par2.o )
Linking sudoku-par2 ...
$
```

Run it on 2 processors

```
> ./sudoku-par2 sudoku17.1000.txt +RTS -s -N2
./sudoku-par2 sudoku17.1000.txt +RTS -s -N2
1000
   2,400,207,256 bytes allocated in the heap
      49,191,144 bytes copied during GC
       2,669,416 bytes maximum residency (7 sample(s))
         339,904 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

  …
  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.24s  (  1.79s elapsed)
  GC    time    1.11s  (  0.20s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.34s  (  1.99s elapsed)
…
```




sudoku-par2.eventlog - ThreadScope
File View Help
Key Traces Bookmarks
running
GC
create thread
run spark
thread runnable
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
Timeline
0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1s 1.1s 1.2s 1.3s 1.4s 1.5s 1.6s 1.7s 1.8s 1.9s 2s
Activity
HEC 0
HEC 1
Events
0.001139s startup: 2 capabilities
0.001453s cap 1: creating thread 1
0.001454s cap 1: thread 1 is runnable
0.001457s cap 1: running thread 1
0.001564s cap 1: stopping thread 1 (making a foreign call)
0.001566s cap 1: running thread 1
0.001573s cap 1: stopping thread 1 (making a foreign call)
sudoku-par2.eventlog (86565 events, 2.081s)

parMap

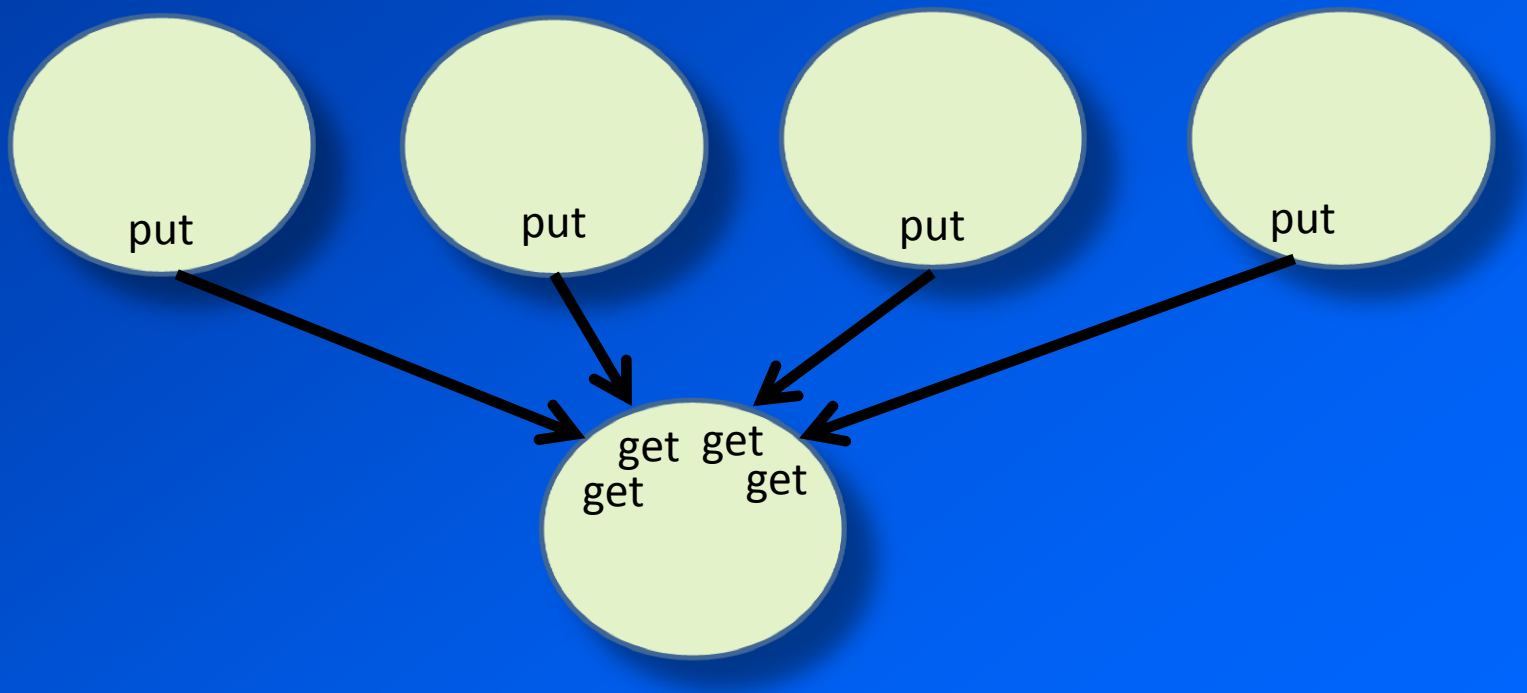
- Let's use the Par monad to define the parMap pattern. First expand our vocabulary a bit:

```
spawn :: Par a -> Par (IVar a)
spawn p = do r <- new
             fork $ p >>= put r
             return r
```

- now define parMap:

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

What is the dataflow graph?



Parallel sudoku solver version 3

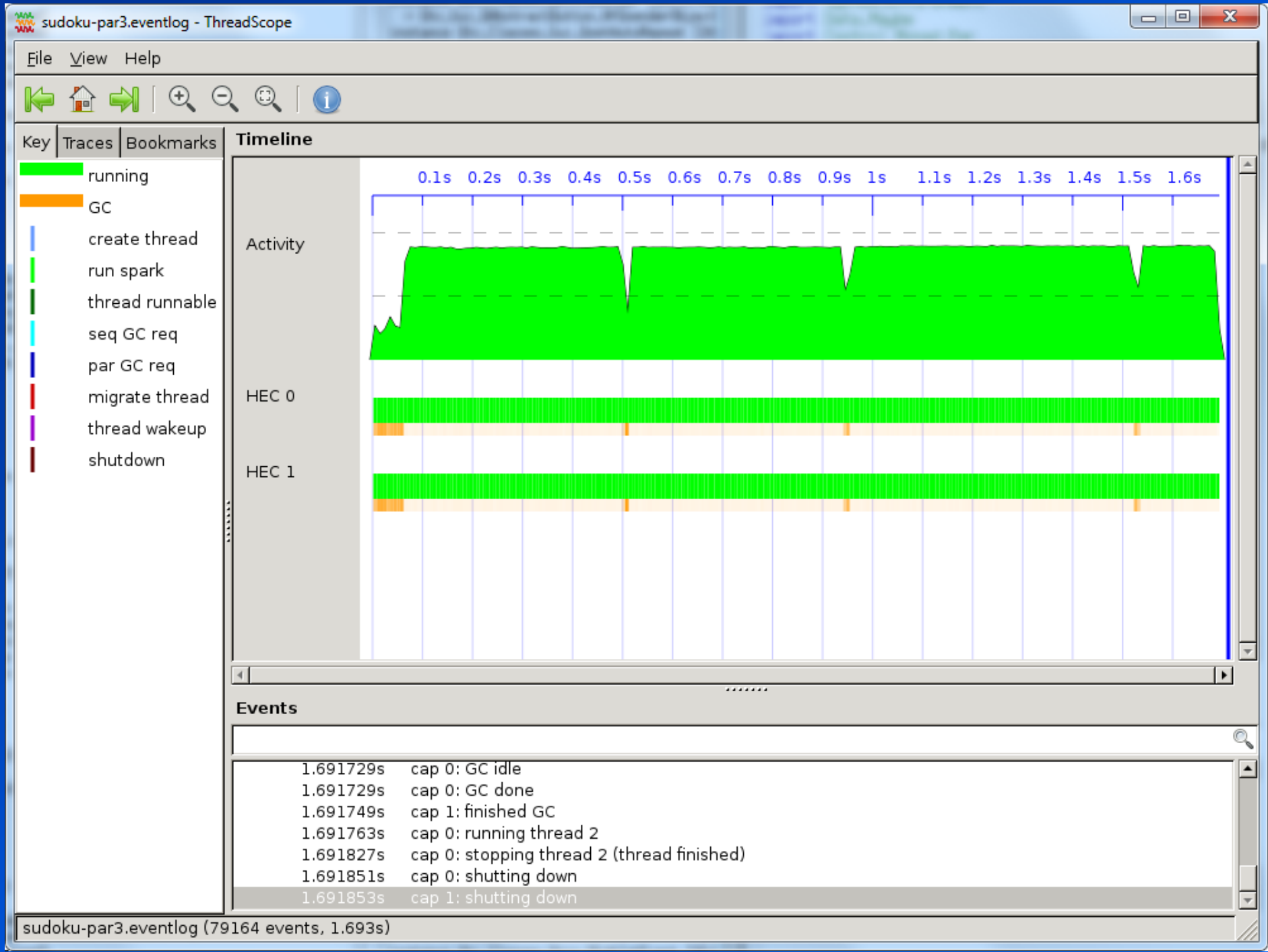
```
main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    print $ length $ filter isJust $
       runPar $ parMap solve grids
```

- How does it perform?

sudoku-par3 on 2 cores

```
./sudoku-par3 sudoku17.1000.txt +RTS -N2 -s
1000
   2,400,973,624 bytes allocated in the heap
      50,751,248 bytes copied during GC
       2,654,008 bytes maximum residency (6 sample(s))
         368,256 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)
…

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.06s  (  1.47s elapsed)
  GC    time    1.29s  (  0.21s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.36s  (  1.68s elapsed)
```

- Speedup: 3.02/1.68 = 1.79


sudoku-par3.eventlog - ThreadScope
File View Help
Key Traces Bookmarks
running
GC
create thread
run spark
thread runnable
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
Timeline
0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1s 1.1s 1.2s 1.3s 1.4s 1.5s 1.6s
Activity
HEC 0
HEC 1
Events
1.691729s cap 0: GC idle
1.691729s cap 0: GC done
1.691749s cap 1: finished GC
1.691763s cap 0: running thread 2
1.691827s cap 0: stopping thread 2 (thread finished)
1.691851s cap 0: shutting down
1.691853s cap 1: shutting down
sudoku-par3.eventlog (79164 events, 1.693s)

Granularity

- Granularity = size of the tasks
  - Too small, and the overhead of fork/get/put will outweigh the benefits of parallelism
  - Too large, and we risk underutilisation (see sudoku-par2.hs)
  - The range of “just right” is often quite wide
- Let’s test that. How do we change the granularity?

parMap with variable granularity

```
parMapChunk :: NFData b => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs = do
  xss <- parMap (map f) (chunk n xs)
  return (concat xss)

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = as : chunk n bs
  where (as,bs) = splitAt n xs
```

- split the list into chunks of size *n*
- Each node processes *n* elements
- (this isn't in the library, but it should be)

Final version of sudoku: chunking

- sudoku-par4.hs

```
main :: IO ()
main = do
    [f,n] <- getArgs
    grids <- fmap lines $ readFile f
    print $ length $ filter isJust $
        runPar $ parMapChunk (read n) solve grids
```

Results with sudoku17.16000.txt

No chunks (sudoku-par3):
  Total time   43.71s (43.73s elapsed)
chunk 100, -N1:
  Total time   44.43s (44.44s elapsed)

No chunks, -N8:
  Total time   67.73s (8.38s elapsed)
  (5.21x)
chunk 10, -N8:
  Total time   61.62s (7.74s elapsed)
  (5.64x)
chunk 100, -N8:
  Total time   60.81s (7.73s elapsed)
  (5.65x)
chunk 1000, -N8:
  Total time   61.74s (7.88s elapsed)
  (5.54x)

Granularity: conclusion

- Use parListChunk if your tasks are too small
- If your tasks are too large, look for ways to divide the work and add more parallelism
- If the number of tasks is less than 10 times the number of cores, that is probably too few

Enough about sudoku!

- We’ve been dealing with flat parallelism so far
- What about other common patterns, such as divide and conquer?

Examples

- Divide and conquer parallelism:

```
parfib :: Int -> Int -> Par Int
parfib n
  | n <= 2    = return 1
  | otherwise = do
     x <- spawn $ parfib (n-1)
     y <- spawn $ parfib (n-2)
     x' <- get x
     y' <- get y
     return (x' + y')
```

Note…

Note…

- We have to thread the Par monad to all the sites we might want to spawn or fork.

Note…

- We have to thread the Par monad to all the sites we might want to spawn or fork.
- Why? Couldn't we just call a new runPar each time?

```
runPar :: Par a -> a
```

Note…

- We have to thread the Par monad to all the sites we might want to spawn or fork.
- Why? Couldn't we just call a new runPar each time?

```
runPar :: Par a -> a
```

- Each runPar:
  - Waits for all its subtasks to finish before returning (necessary for determinism)
  - Fires up a new gang of N threads and creates scheduling data structures: it's expensive
  - So we do want to thread the Par monad around

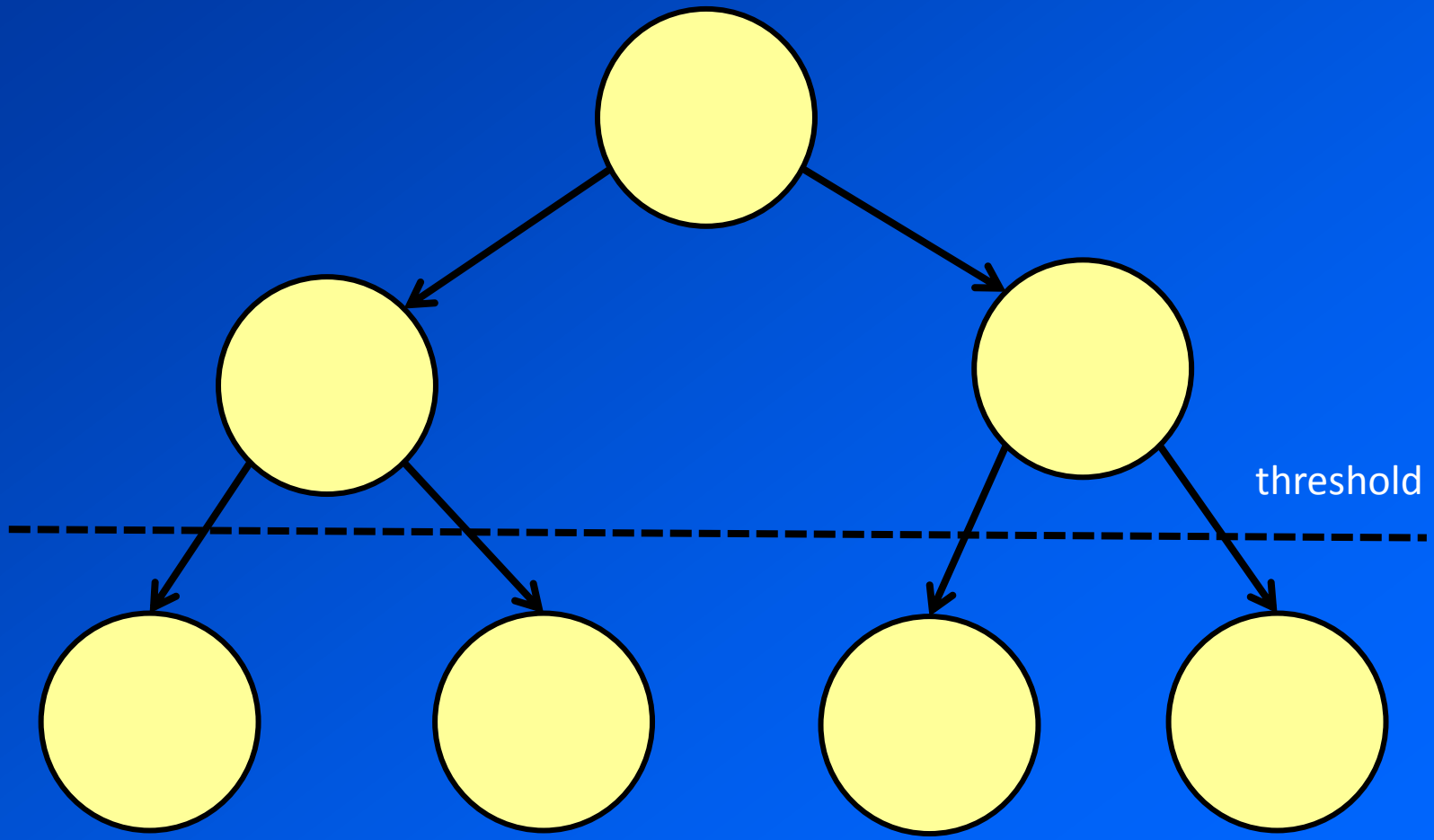
Granularity in divide-and-conquer

- If you try to run this, performance will be terrible:

```
parfib :: Int -> Par Int
parfib n
  | n <= 2    = return 1
  | otherwise = do
     x <- spawn $ parfib (n-1)
     y <- spawn $ parfib (n-2)
     x' <- get x
     y' <- get y
     return (x' + y')
```

- For a start, it's 50x slower than the sequential version
  - overhead of the Par monad

- As we saw before, when our tasks are too small we need to increase the granularity
- Here there's no obvious place to do chunking
- Instead we want to set a threshold for task creation



- parfib takes an extra parameter, the threshold
- below the threshold, we use the sequential fib
- a threshold of e.g. 25 is enough to give almost perfect speedup

```
parfib :: Int -> Int -> Par Int
parfib n t
  | n <= 2    = return 1
  | n <= t    = fib n
  | otherwise = do
     x <- spawn $ parfib (n-1) t
     y <- spawn $ parfib (n-2) t
     x' <- get x
     y' <- get y
     return (x' + y')

fib :: Int -> Int
fib n = ...
```

- parfib isn't a very realistic example
- let's try sorting instead
- mergesort:
  - divide the list into two
  - sort each half
  - merge the results

```
parsort :: Int -> [Integer] -> Par [Integer]
parsort n []  = return []
parsort n [x] = return [x]
parsort 0 xs  = return (sort xs)
parsort n xs  = do
  let (as,bs) = split xs
  l <- spawn $ parsort (n-1) as
  r <- spawn $ parsort (n-1) bs
  ls <- get l
  rs <- get r
  return (merge ls rs)
```

- n is the threshold
- sort is the sequential sort

Skeletons

- Parallelism often fits a well-known pattern
- We've seen two common patterns so far:
  - parallel map
  - divide-and-conquer
- The idea of a skeleton is to abstract the pattern as a reusable higher-order function
- parMap is already a skeleton

Divide and conquer as a skeleton

```
divConq :: NFData sol
        => (prob -> Bool)          -- indivisible?
        -> (prob -> (prob,prob)) -- split into subproblems
        -> (sol -> sol -> sol)   -- join solutions
        -> (prob -> sol)           -- solve a subproblem
        -> (prob -> sol)

divConq indiv split join f prob
 = runPar $ go prob
 where
   go prob
     | indiv prob = return (f prob)
     | otherwise = do
         let (a,b) = split prob
         i <- spawn $ go a
         j <- spawn $ go b
         a <- get i
         b <- get j
         return (join a b)
```

- Using the skeleton
- Our “prob” is (Int,[Integer])
  - i.e. pair the threshold counter with the list

```
parsort :: Int -> [Integer] -> [Integer]
parsort thresh xs
  = divConq indiv divide merge (sort . snd) (thresh,xs)
  where
    indiv (n,xs) = n == 0

    divide (n,xs) = ((n-1, as), (n-1, bs))
      where (as,bs) = split xs
```

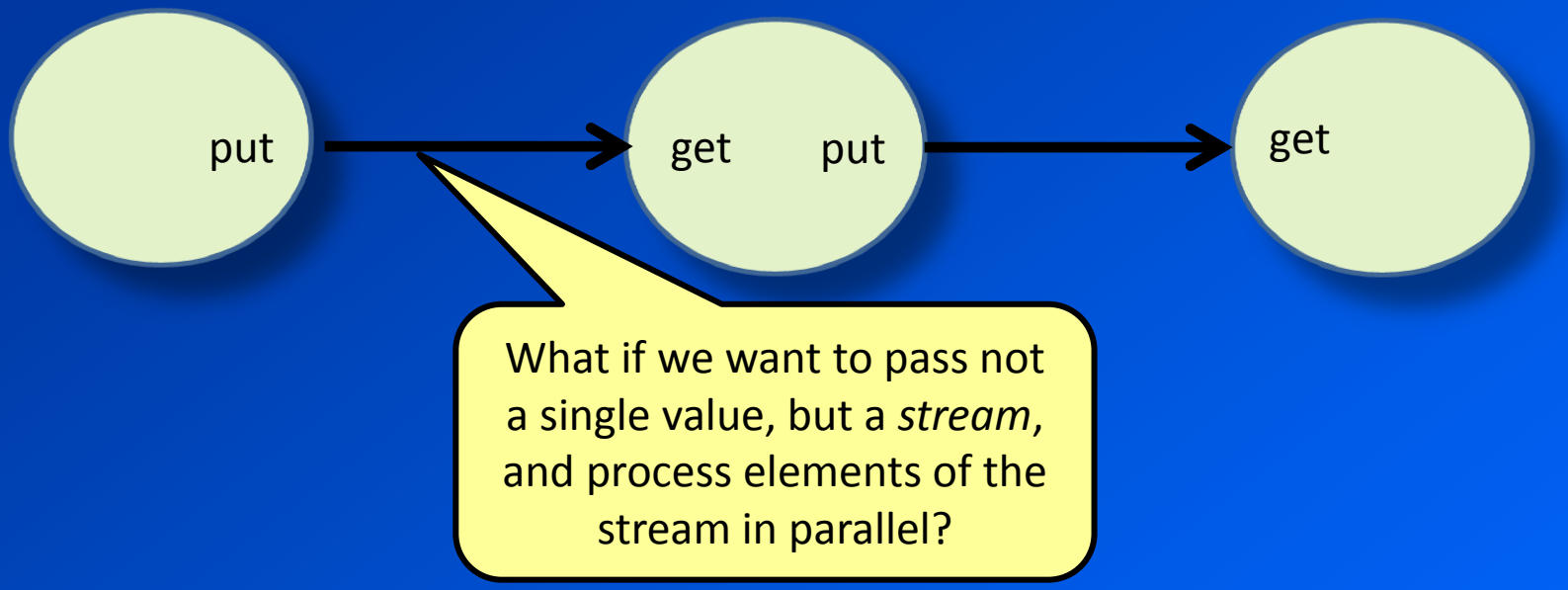
- Nice compact definition of parallel sorting
- Important: the details of the parallelism are hidden in divConq (we could have used Strategies)

Rule of thumb

- Try to separate the *application code* from the *parallel coordination* by using higher-order skeletons
- Good abstraction facilities lead to modularity

break...

Pipeline parallelism



Or to put it another way:

- if we have a computation that produces a list
- and another one that consumes it
- how can we overlap these with the Par monad?

IList and Stream

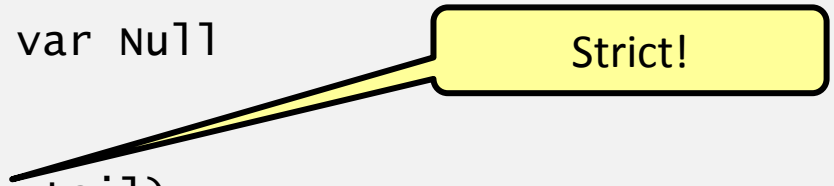
```
data IList a = Null
             | Cons { hd :: a
                    , tl :: Stream a }

type Stream a = IVar (IList a)
```

- Stream is a “lazy list” in the Par monad
  - but we’re being explicit about where the laziness is
- We need a way to:
  - Generate a new Stream
  - Process a stream (map, filter)
  - Consume a Stream (fold)
- Plugging these together gives us parallel pipeline processing.
- Stream code is in code/euler35/Stream.hs

Generate a Stream

- One way: generate a stream from a real lazy list:

```
fromList :: NFData a => [a] -> Par (Stream a)
fromList xs =
   do var <- new
      fork $ loop xs var
      return var
 where
  loop [] var  =  put var Null
  loop (x:xs) var =
    do tail <- new
       put var (Cons x tail)
       loop xs tail
```



Filter a Stream

```
streamFilter :: NFData a => (a -> Bool) -> Stream a
             -> Par (Stream a)
streamFilter p instr = do
    outstr <- new
    fork $ loop instr outstr
    return outstr
  where
    loop instr outstr = do
      v <- get instr
      case v of
        Null -> put outstr Null
        Cons x instr'
          | p x -> do
             tail <- new
             put_ outstr (Cons x tail)
             loop instr' tail
          | otherwise -> do
             loop instr' outstr
```

Consume a stream

- Analogue of foldl:

```
streamFold :: (a -> b -> a) -> a -> Stream b -> Par a
streamFold fn acc instrm =
   do ilst <- get instrm
      case ilst of
        Null     -> return acc
        Cons h t -> streamFold fn (fn acc h) t
```

- This version is not strict – maybe it should be?

Pipeline example

- Project Euler problem 35: “Find all the *circular* primes below 1,000,000”. A circular prime is one in which all rotations of its digits are also prime.

```
main :: IO ()
main = print $ runPar $ do
     s1 <- streamFromList (takeWhile (<1000000) primes)
     s2 <- streamFilter circular s1
     streamFold (\a _ -> a + 1) 0 s2
```

- Full code is in code/euler35/euler35.hs
- Achieves 1.85 speedup vs. the sequential version on 2 cores (does not scale further)
- Beware: this is not suitable for working with streams that do not fit in memory, since there is nothing preventing the producer from producing elements too fast

Dataflow problems

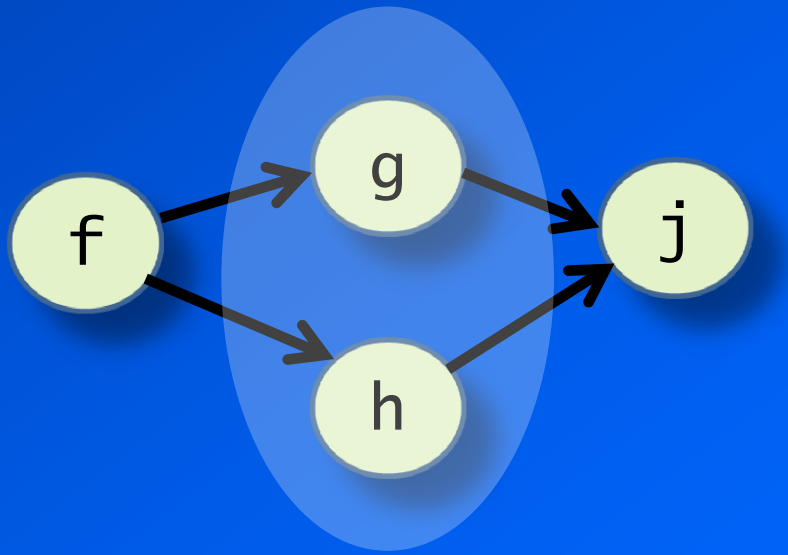
- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

Example

- Consider typechecking a functional program
- A set of bindings of the form x = e
- To typecheck a binding:
  - input: the types of the variables 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

Example

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



Parallel

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
...
```

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
```

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
```

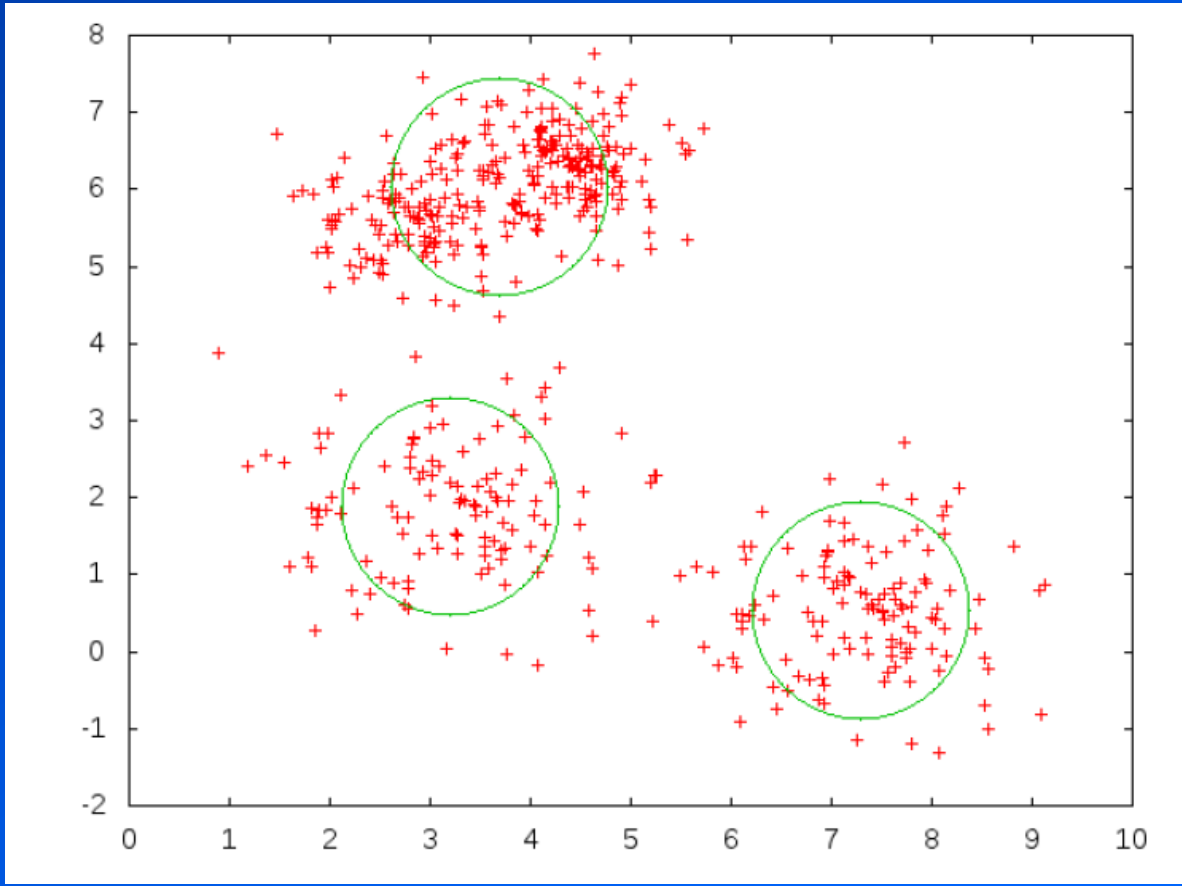
Results

```
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

K-Means

- 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!

K-Means: basics

```
data Vector = Vector Double Double

addVector :: Vector -> Vector -> Vector
addVector (Vector a b) (Vector c d) = Vector (a+c) (b+d)

data Cluster = Cluster
               {
                  clId    :: !Int,
                  clCount :: !Int,
                  clSum   :: !Vector,
                  clCent  :: !Vector
               }

sqDistance :: Vector -> Vector -> Double
 -- square of distance between vectors

makeCluster :: Int -> [Vector] -> Cluster
 -- builds Cluster from a set of points
```

K-Means:

```
assign
  :: Int                    -- number of clusters
  -> [Cluster]              -- clusters
  -> [Vector]               -- points
  -> Array Int [Vector]     -- points assigned to clusters

makeNewClusters :: Array Int [Vector] -> [Cluster]
 -- takes result of assign, produces new clusters

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

- assign is step 2 (assign points to clusters)
- makeNewClusters is step 3 (compute average of points to get new clusters)
- step is (2,3) – one iteration

Putting it together.. sequentially

```
kmeans_seq :: Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_seq nclusters points clusters = do
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany = 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
```

Parallelise makeNewClusters?

```
makeNewClusters :: Array Int [Vector] -> [Cluster]
makeNewClusters arr =
  filter ((>0) . clCount) $
     [ makeCluster i ps | (i,ps) <- assocs arr ]
```

- essentially a map over the clusters
- number of clusters is small
- not enough parallelism here – grains are too large, fan-out is too small

How to parallelise?

- Parallelise assign?

```
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 = ...
```

- essentially map/reduce: map nearest + accumArray to attach each point to its cluster
- the map parallelises, but accumArray doesn't
- could divide into chunks... but is there a better way?

Sub-divide the data

- Suppose we divided the data set in two, and called step on each half
- We would need a way to combine the results:

```
step n cs (as ++ bs) == step n cs as `combine` step n cs bs
```

- but what is combine?

```
combine :: [Cluster] -> [Cluster] -> [Cluster]
```

- assuming we can match up cluster pairs, we just need a way to combine two clusters

Combining clusters

- A cluster is notionally a set of points
- Its *centroid* is the average of the points
- A Cluster is represented by its centroid:

```
data Cluster = Cluster
               {
                  clId    :: !Int,
                  clCount :: !Int,      -- num of points
                  clSum   :: !Vector,   -- sum of points
                  clCent  :: !Vector    -- clSum / clCount
               }
```

- but note that we cached clCount and clSum
- these let us merge two clusters and recompute the centroid in O(1)

Combining clusters

- So using

```
combineClusters :: Cluster -> Cluster -> Cluster
```

- we can define

```
reduce :: Int -> [[Cluster]] -> [Cluster]
```

- (see notes for the code; straightforward)
- now we can express K-Means as a map/reduce

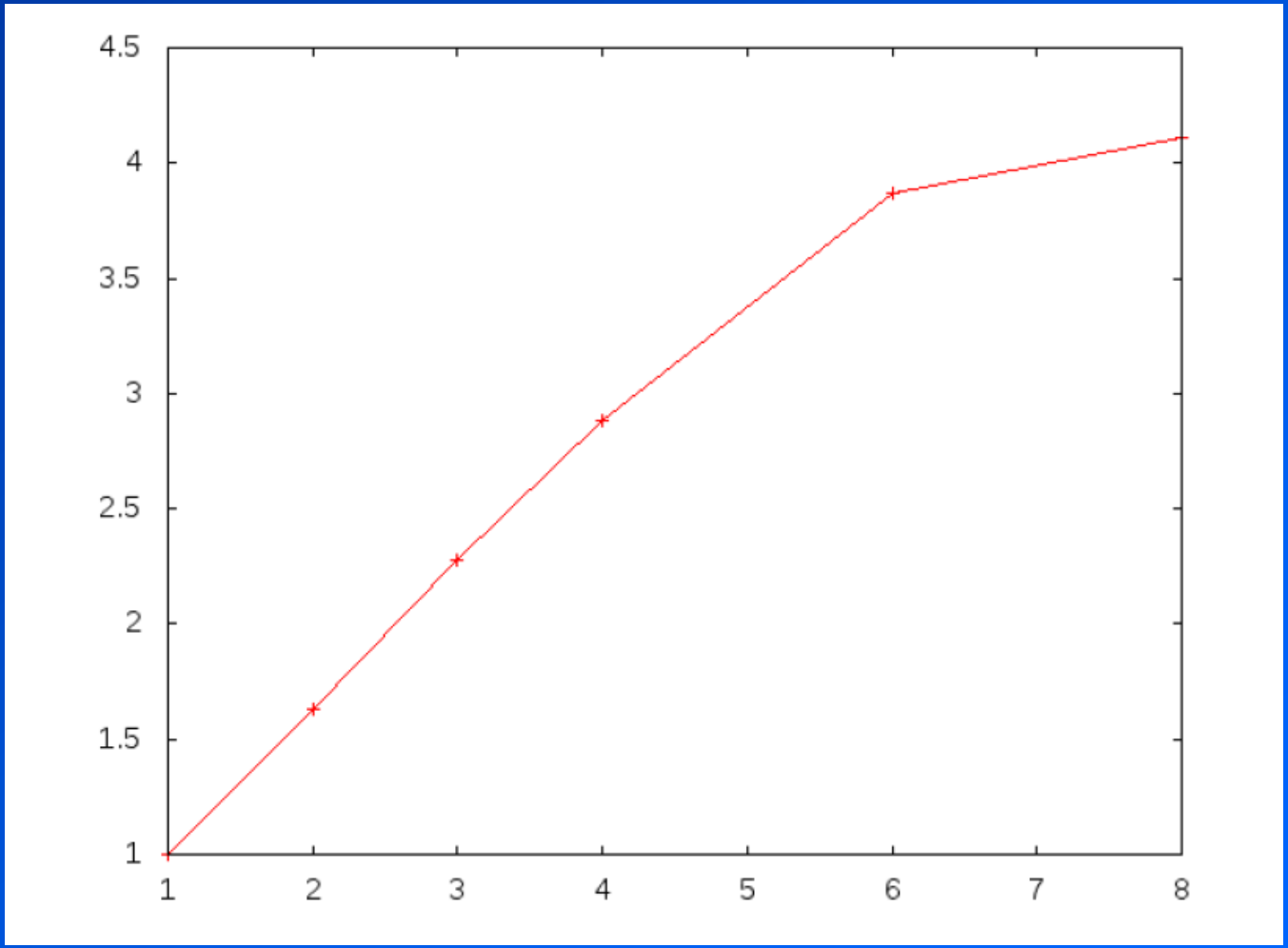
Final parallel implementation

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

             clusters' = reduce nclusters new_clusterss

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

What chunk size?

- Divide data by number of processors?
  - No! Static partitioning could lead to poor utilisation (see earlier)
  - there's no need to have such large chunks, the RTS will schedule smaller work items across the available cores

- Results for 170000 2-D points, 4 clusters, 1000 chunks



Further thoughts

- We had to restructure the algorithm to make the maximum amount of parallelism available
  - map/reduce
  - move the branching point to the top
  - make reduce as cheap as possible
  - a tree of reducers is also possible
- Note that the parallel algorithm is data-local – this makes it particularly suitable for distributed parallelism (indeed K-Means is commonly used as an example of distributed parallelism).
- But be careful of static partitioning

Thoughts to take away...

Thoughts to take away...

- Making your program faster is the goal
  - Parallelism is just one way to achieve that
    - it might not be the easiest way!
  - However, designing your code with parallelism in mind should ensure that it can ride Moore's law a bit longer
  - good:
    - maps and trees
  - suspicious:
    - folds (but associative folds are OK)
    - lists (operations on lists themselves are serial, but operations on the elements of a list can be parallelised)

Exercises

- Don’t use the printout!
- http://community.haskell.org/~simonmar/lab-exercises-cadarache.pdf
- includes instructions for downloading the sample code
- Lab 1 covers the first two lectures
- Enjoy!