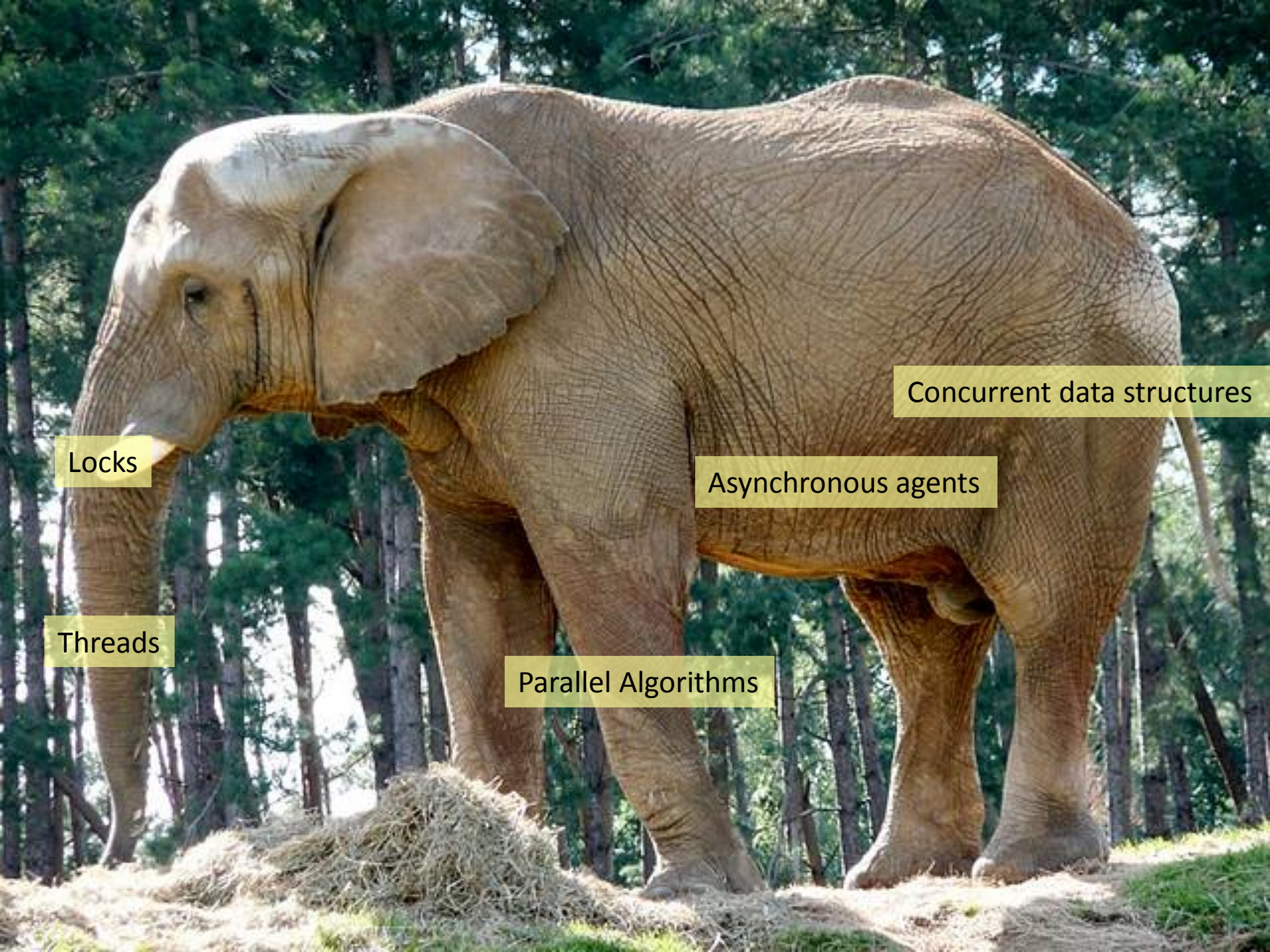
Parallel and Concurrent Haskell Part I

Simon Marlow
(Microsoft Research, Cambridge, UK)

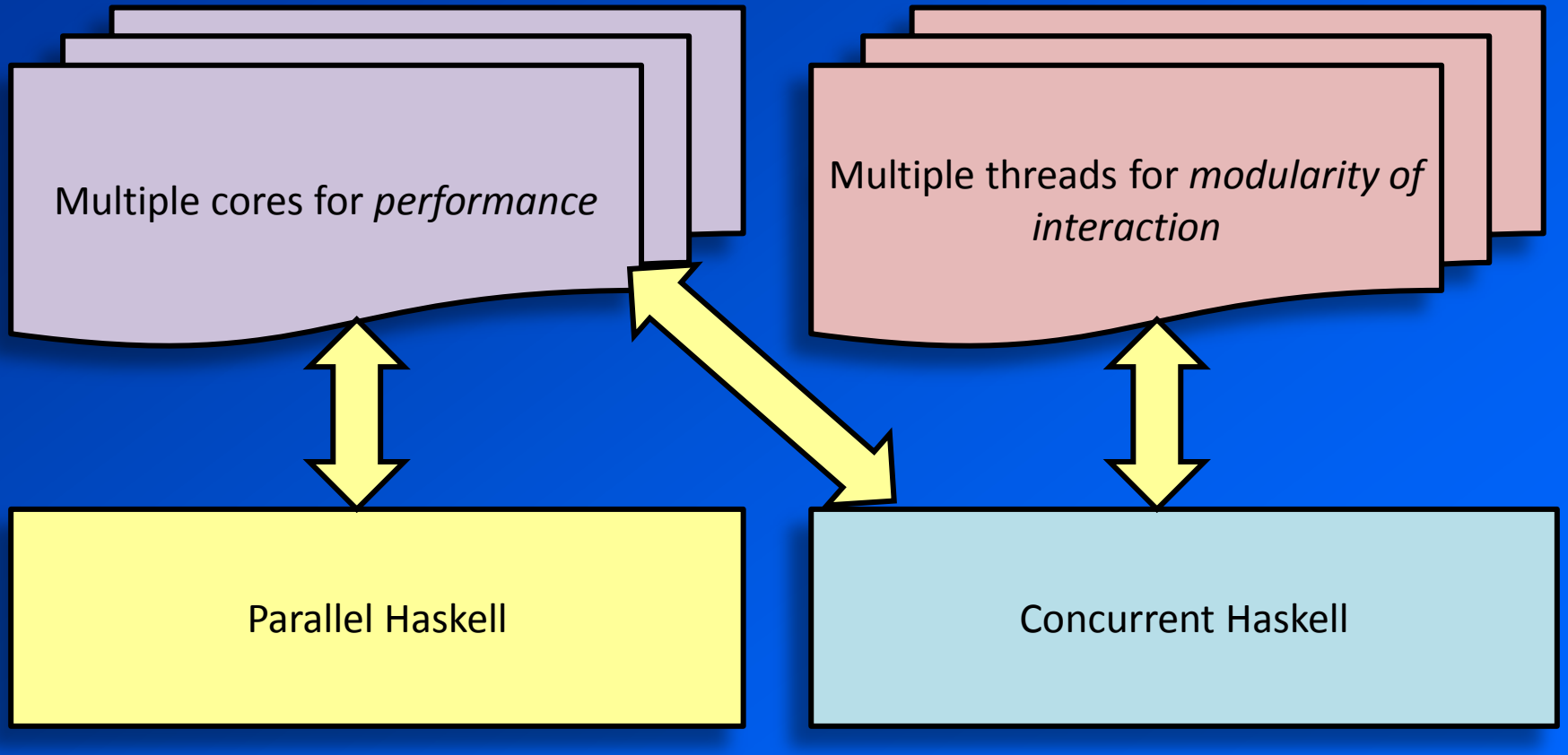

Concurrent data structures
Locks
Asynchronous agents
Threads
Parallel Algorithms

All you need is *X*

- Where *X* is actors, threads, transactional memory, futures...
- Often true, but for a given application, some *X*s will be much more suitable than others.
- In Haskell, our approach is to give you lots of different *X*s
  - "Embrace diversity (but control side effects)" (Simon Peyton Jones)

Parallel and Concurrent Haskell ecosystem

Parallelism vs. Concurrency



Parallelism vs. Concurrency

- Primary distinguishing feature of Parallel Haskell: determinism
  - The program does “the same thing” regardless of how many cores are used to run it.
  - No race conditions or deadlocks
  - add parallelism without sacrificing correctness
  - Parallelism is used to speed up pure (non-IO monad) Haskell code

Parallelism vs. Concurrency

- Primary distinguishing feature of Concurrent Haskell: threads of control
  - Concurrent programming is done in the IO monad
    - because threads have *effects*
    - effects from multiple threads are interleaved nondeterministically at runtime.
  - Concurrent programming allows programs that interact with multiple external agents to be *modular*
    - the interaction with each agent is programmed separately
    - Allows programs to be structured as a collection of interacting agents (actors)

I. Parallel Haskell

- In this part of the course, you will learn how to:
  - Do basic parallelism:
    - compile and run a Haskell program, and measure its performance
    - parallelise a simple Haskell program (a Sudoku solver)
    - use ThreadScope to profile parallel execution
    - do dynamic partitioning
    - measure parallel speedup
      - use Amdahl's law to calculate possible speedup
  - Work with Evaluation Strategies
    - build simple Strategies
    - parallelise a data-mining problem: K-Means
  - Work with the Par Monad
    - Use the Par monad for expressing dataflow parallelism
    - Parallelise a type-inference engine

Running example: solving Sudoku

- code from the Haskell wiki (brute force search with some intelligent pruning)
- can solve all 49,000 problems in 2 mins
- input: a line of text representing a problem

```
.......2143.......6........2.15..........637...........68...4.....23........7....
.......241..8.............3...4..5..7.....1......3.......51.6....2....5..3...7...
.......24....1...........8.3.7...1..1..8..5.....2......2.4...6.5...7.3...........
```

```
import Sudoku

solve :: String -> Maybe Grid
```

Solving Sudoku problems

- Sequentially:
  - divide the file into lines
  - call the solver for each line

```
import Sudoku
import Control.Exception
import System.Environment

main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    mapM (evaluate . solve) grids
```

```
evaluate :: a -> IO a
```

Compile the program...

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$
```

Run the program...

```
$ ./sudoku1 sudoku17.1000.txt +RTS -s
   2,392,127,440 bytes allocated in the heap
      36,829,592 bytes copied during GC
         191,168 bytes maximum residency (11 sample(s))
          82,256 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  4570 collections,     0 parallel,  0.14s,  0.13s elapsed
  Generation 1:    11 collections,     0 parallel,  0.00s,  0.00s elapsed

...

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.92s  (  2.92s elapsed)
  GC    time    0.14s  (  0.14s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.06s  (  3.06s elapsed)

...
```

Now to parallelise it...

- Doing parallel computation entails specifying coordination in some way – compute A in parallel with B
- This is a constraint on evaluation order
- But by design, Haskell *does not have a specified evaluation order*
- So we need to add something to the language to express constraints on evaluation order

The Eval monad

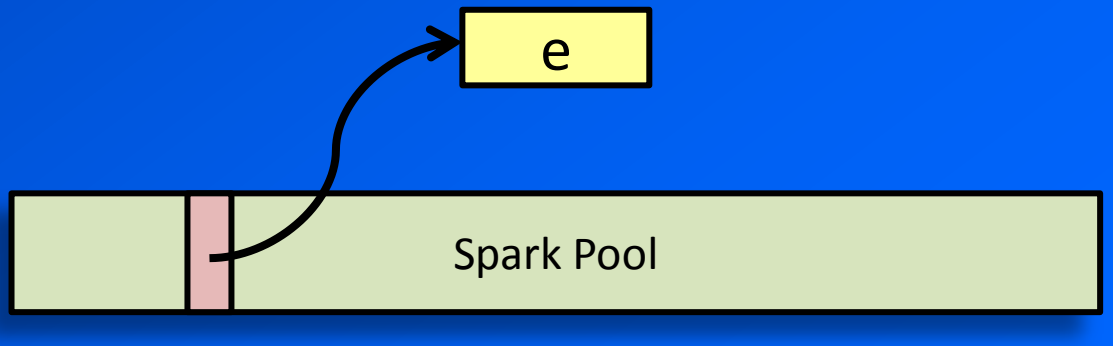
```
import Control.Parallel.Strategies

data Eval a
instance Monad Eval

runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

- Eval is pure
- Just for expressing sequencing between rpar/rseq – nothing more
- Compositional – larger Eval sequences can be built by composing smaller ones using monad combinators
- Internal workings of Eval are very simple (see Haskell Symposium 2010 paper)

What does rpar *actually do?*

```
x <- rpar e
```

- rpar creates a *spark* by writing an entry in the *spark pool*
  - rpar is very cheap! (not a thread)
- the spark pool is a circular buffer
- when a processor has nothing to do, it tries to remove an entry from its own spark pool, or steal an entry from another spark pool (*work stealing)*
- when a spark is found, it is evaluated
- The spark pool can be full – watch out for spark overflow!



Basic Eval patterns

- To compute a in parallel with b, and return a pair of the results:

```
do
   a' <- rpar a
   b' <- rseq b
   return (a',b')
```

Start evaluating a in the background

Evaluate b, and wait for the result

- alternatively:

```
do
   a' <- rpar a
   b' <- rseq b
   rseq a'
   return (a',b')
```

- what is the difference between the two?

Parallelising Sudoku

- Let's divide the work in two, so we can solve each half in parallel:

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

- Now we need something like

```
runEval $ do
   as' <- rpar (map solve as)
   bs' <- rpar (map solve bs)
   rseq as'
   rseq bs'
   return ()
```

But this won't work...

```
runEval $ do
   as' <- rpar (map solve as)
   bs' <- rpar (map solve bs)
   rseq as'
   rseq bs'
   return ()
```

- rpar evaluates its argument to Weak Head Normal Form (WHNF)
- WTF is WHNF?
  - evaluates as far as the *first constructor*
  - e.g. for a list, we get either [] or (x:xs)
  - e.g. WHNF of "map solve (a:as)" would be "solve a : map solve as"
- But we want to evaluate the whole list, and the elements

We need to go deeper

```
import Control.DeepSeq
deep :: NFData a => a -> a
deep a = deepseq a a
```

- deep fully evaluates a nested data structure and returns it
  - e.g. a list: the list is fully evaluated, including the elements
- uses overloading: the argument must be an instance of NFData
  - instances for most common types are provided by the library

Ok, adding deep

```
runEval $ do
   as' <- rpar (deep (map solve as))
   bs' <- rpar (deep (map solve bs))
   rseq as'
   rseq bs'
   return ()
```

- Now we just need to evaluate this at the top level in 'main':

```
evaluate $ runEval $ do
      a <- rpar (deep (map solve as))
      ...
```

- (normally using the result would be enough to force evaluation, but we're not using the result here)

Let’s try it...

- Compile sudoku2
  - (add -threaded -rtsopts)
  - run with `sudoku17.1000.txt +RTS -N2`
- Take note of the Elapsed Time

Runtime results...

```
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s
   2,400,125,664 bytes allocated in the heap
      48,845,008 bytes copied during GC
       2,617,120 bytes maximum residency (7 sample(s))
         313,496 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  2975 collections,  2974 parallel,  1.04s,  0.15s elapsed
  Generation 1:     7 collections,     7 parallel,  0.05s,  0.02s elapsed

  Parallel GC work balance: 1.52 (6087267 / 3999565, ideal 2)

  SPARKS: 2 (1 converted, 0 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.21s  (  1.80s elapsed)
  GC    time    1.08s  (  0.17s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.29s  (  1.97s elapsed)
```

Calculating Speedup

- Calculating speedup with 2 processors:
  - Elapsed time (1 proc) / Elapsed Time (2 procs)
  - NB. not CPU time (2 procs) / Elapsed (2 procs)!
  - NB. compare against sequential program, not parallel program running on 1 proc

- Speedup for sudoku2: 3.06/1.97 = 1.55
  - not great...

Why not 2?

- there are two reasons for lack of parallel speedup:
  - less than 100% utilisation (some processors idle for part of the time)
  - extra overhead in the parallel version
- Each of these has many possible causes...

A menu of ways to screw up

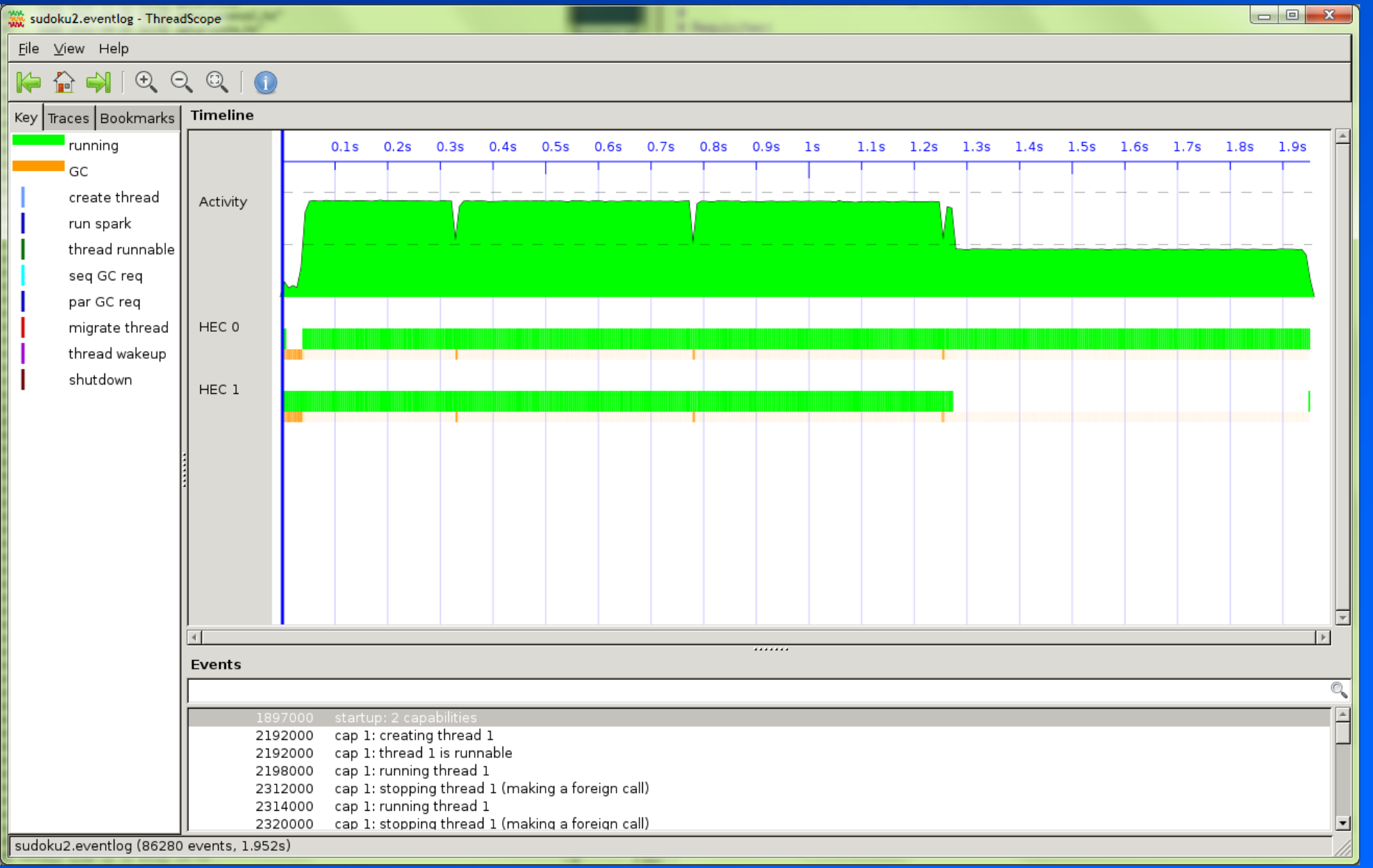
- less than 100% utilisation
  - parallelism was not created, or was discarded
  - algorithm not fully parallelised – residual sequential computation
  - uneven work loads
  - poor scheduling
  - communication latency
- extra overhead in the parallel version
  - overheads from rpar, work-stealing, deep, ...
  - lack of locality, cache effects...
  - larger memory requirements leads to GC overhead
  - GC synchronisation
  - duplicating work

So we need *tools*

- to tell us why the program isn't performing as well as it could be
- For Parallel Haskell we have ThreadScope

```
$ rm sudoku2; ghc -O2 sudoku2.hs -threaded -rtsopts -eventlog
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -ls
$ threadscope sudoku2.eventlog
```

- -eventlog has very little effect on runtime
  - important for profiling parallelism


sudoku2.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
Activity
HEC 0
HEC 1
Events
1897000 startup: 2 capabilities
2192000 cap 1: creating thread 1
2192000 cap 1: thread 1 is runnable
2198000 cap 1: running thread 1
2312000 cap 1: stopping thread 1 (making a foreign call)
2314000 cap 1: running thread 1
2320000 cap 1: stopping thread 1 (making a foreign call)
sudoku2.eventlog (86280 events, 1.952s)

Uneven workloads...

- So one of the tasks took longer than the other, leading to less than 100% utilisation

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

- One of these lists contains more work than the other, even though they have the same length
  - sudoku solving is not a constant-time task: it is a searching problem, so depends on how quickly the search finds the solution

Partitioning

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

- Dividing up the work along fixed pre-defined boundaries, as we did here, is called *static partitioning*
  - static partitioning is simple, but can lead to under-utilisation if the tasks can vary in size
  - static partitioning does not adapt to varying availability of processors – our solution here can use only 2 processors

Dynamic Partitioning

- Dynamic partitioning involves
  - dividing the work into smaller units
  - assigning work units to processors dynamically at runtime using a *scheduler*
  - good for irregular problems and varying number of procoessors
- GHC's runtime system provides spark pools to track the work units, and a work-stealing scheduler to assign them to processors
- So all we need to do is use smaller tasks and more rpars, and we get dynamic partitioning

Revisiting Sudoku...

- So previously we had this:

```
runEval $ do
        a <- rpar (deep (map solve as))
        b <- rpar (deep (map solve bs))
        ...
```

- We want to push rpar down into the map
  - each call to solve will be a separate spark

A parallel map

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do
   b <- rpar (f a)
   bs <- parMap f as
   return (b:bs)
```

Create a spark to evaluate (f a) for each element a

Return the new list

- Provided by Control.Parallel.Strategies
- Also:

```
parMap f xs = mapM (rpar . f) xs
```

Putting it together...

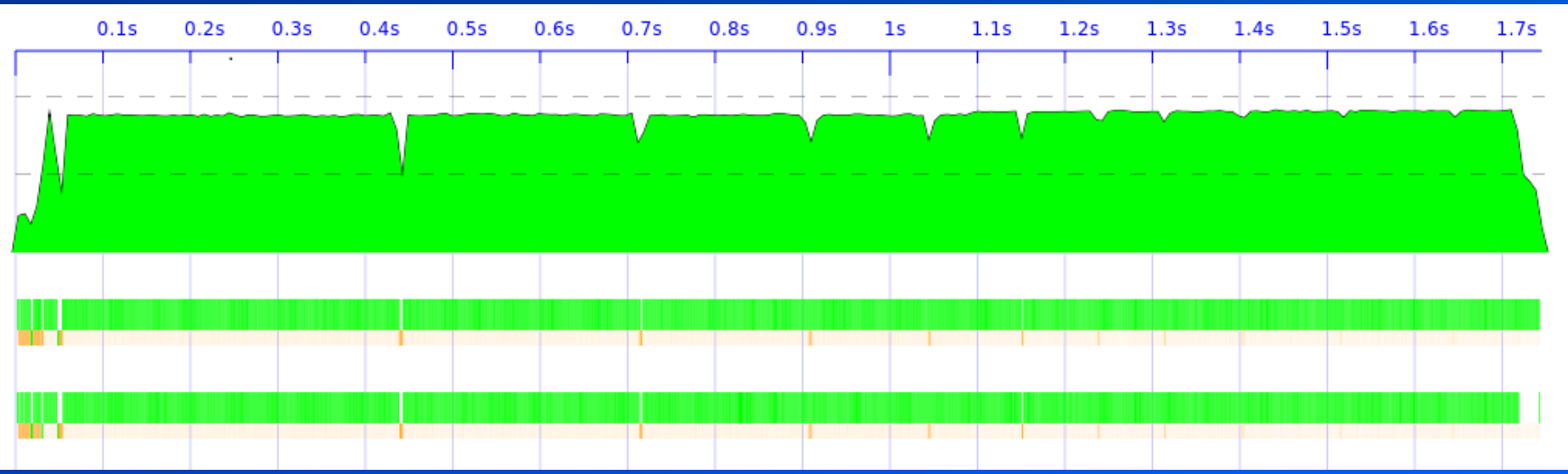
```
evaluate $ deep $ runEval $ parMap solve grids
```

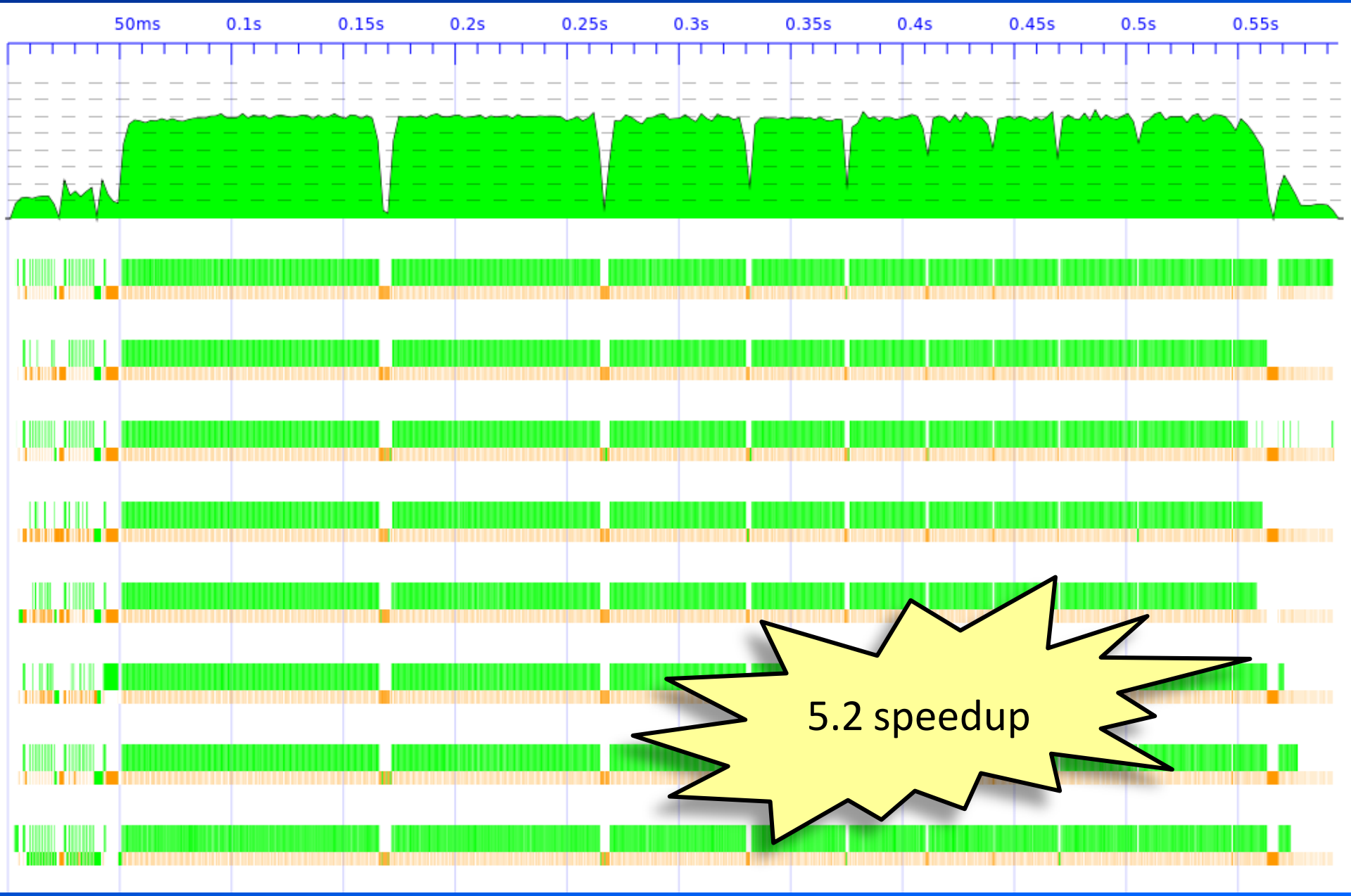
- NB. evaluate $ deep to fully evaluate the result list
- Code is simpler than the static partitioning version!

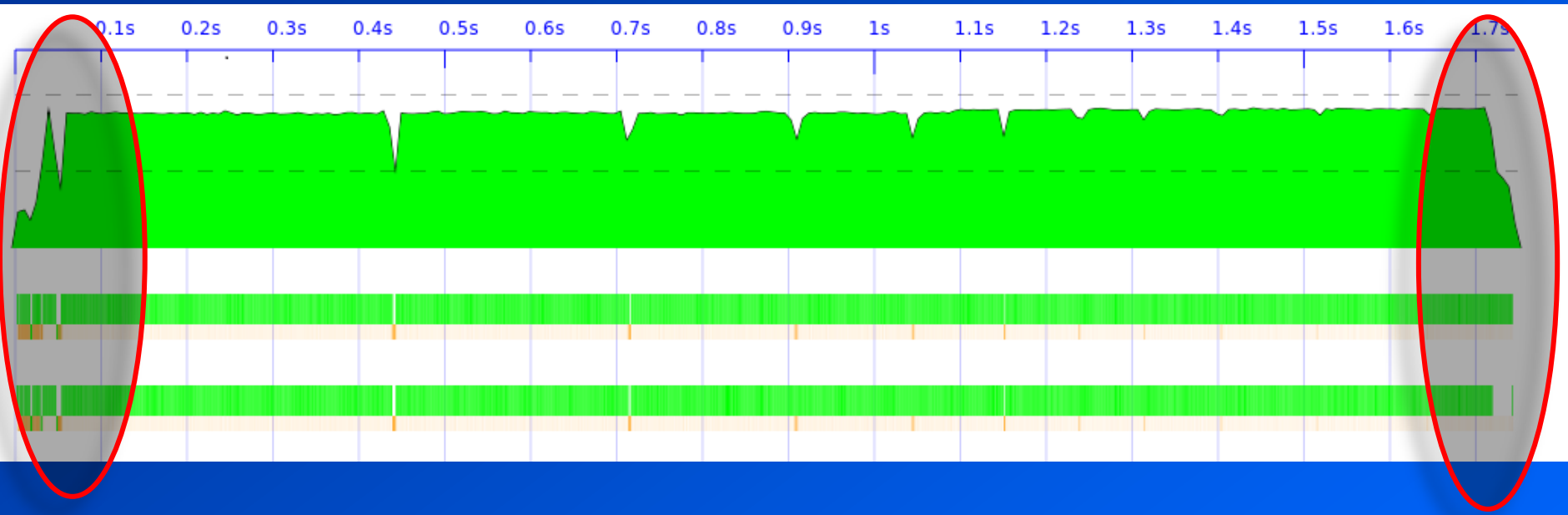
Results

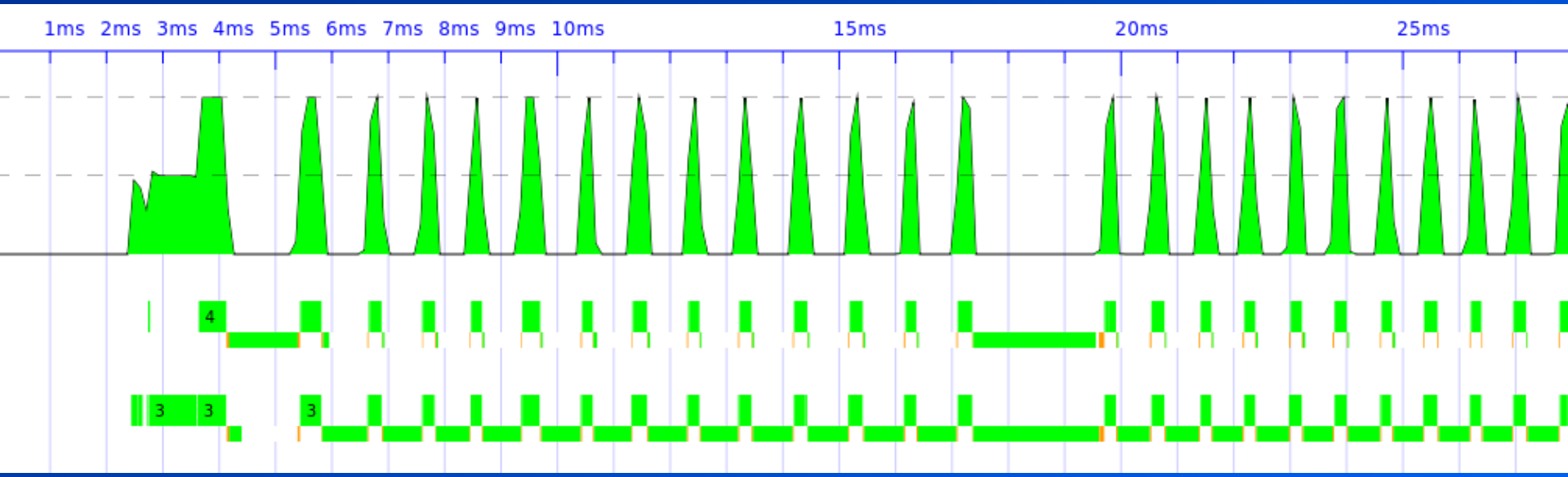
```
./sudoku3 sudoku17.1000.txt +RTS -s -N2 -ls
   2,401,880,544 bytes allocated in the heap
      49,256,128 bytes copied during GC
       2,144,728 bytes maximum residency (13 sample(s))
         198,944 bytes maximum slop
               7 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  2495 collections,  2494 parallel,  1.21s,  0.17s elapsed
  Generation 1:    13 collections,    13 parallel,  0.06s,  0.02s elapsed

  Parallel GC work balance: 1.64 (6139564 / 3750823, ideal 2)

  SPARKS: 1000 (1000 converted, 0 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.19s  (  1.55s elapsed)
  GC    time    1.27s  (  0.19s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.46s  (  1.74s elapsed)
```

Now 1.7 speedup


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


50ms
0.1s
0.15s
0.2s
0.25s
0.3s
0.35s
0.4s
0.45s
0.5s
0.55s
5.2 speedup


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



- Lots of GC
- One core doing all the GC work
  - indicates one core generating lots of data

```
import Sudoku
import Control.Exception
import System.Environment

main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    evaluate $ deep $ runEval $ parMap solve grids
```

- Are there any sequential parts of this program?
- readFile and lines are not parallelised

- Suppose we force the sequential parts to happen first...

```
import Sudoku
import Control.Exception
import System.Environment

main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    evaluate (length grids)
    evaluate $ deep $ runEval $ parMap solve grids
```

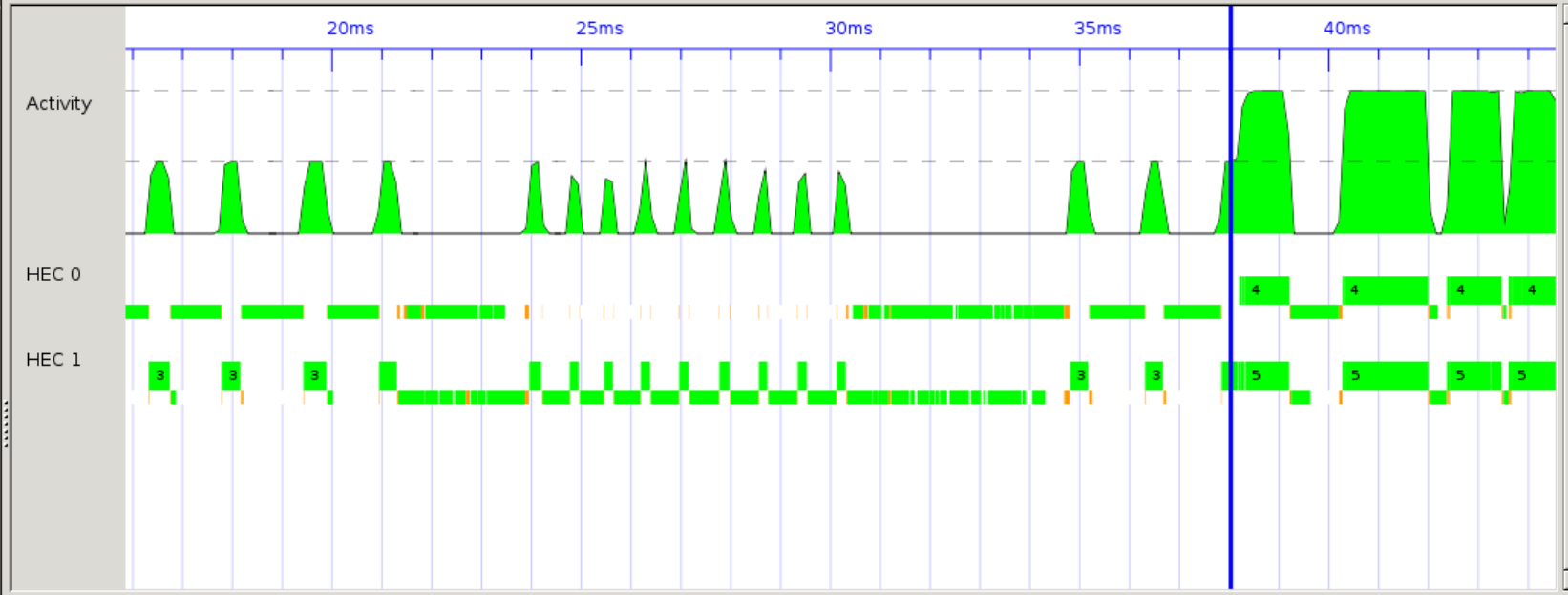
sudoku4.eventlog - ThreadScope

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



Events

38165000	cap 1: stopping thread 3 (stack overflow)
38174000	cap 1: running thread 3
38196000	cap 0: creating thread 4
38197000	cap 0: creating spark thread 4
38197000	cap 0: thread 4 is runnable
38200000	cap 0: running thread 4
38211000	cap 0: thread 4 stealing a spark from cap 1

sudoku4.eventlog (89700 events, 1.800s)

Calculating possible speedup

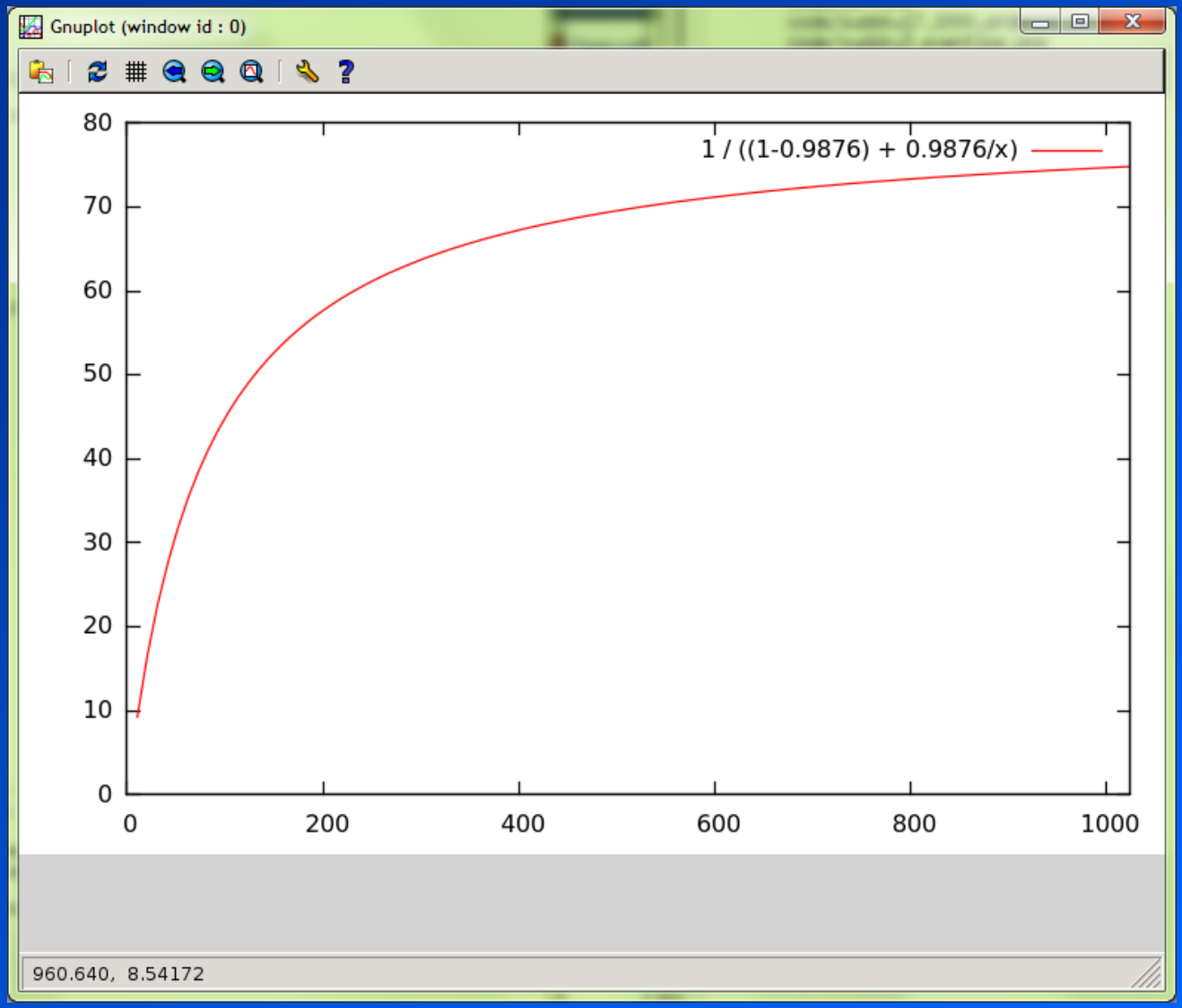
- When part of the program is sequential, Amdahl's law tells us what the maximum speedup is.

$$\frac{1}{(1-P)+\frac{P}{N}}$$

- P = parallel portion of runtime
- N = number of processors

Applying Amdahl's law

- In our case:
  - runtime = 3.06s (NB. sequential runtime!)
  - non-parallel portion = 0.038s (P = 0.9876)
  - N = 2, max speedup = 1 / ((1 – 0.9876) + 0.9876/2)
    - =~ 1.98
    - on 2 processors, maximum speedup is not affected much by this sequential portion
  - N = 64, max speedup = 35.93
    - on 64 processors, 38ms of sequential execution has a dramatic effect on speedup



- diminishing returns...
- See *"Amdahl's Law in the Multicore Era"*, Mark Hill & Michael R. Marty

- Amdahl's law paints a bleak picture
  - speedup gets increasingly hard to achieve as we add more cores
  - returns diminish quickly when more cores are added
  - small amounts of sequential execution have a dramatic effect
  - proposed solutions include heterogeneity in the cores
    - likely to create bigger problems for programmers
- See also Gustafson's law – the situation might not be as bleak as Amdahl's law suggests:
  - with more processors, you can solve a bigger problem
  - the sequential portion is often fixed or grows slowly with problem size
- Note: in Haskell it is hard to identify the sequential parts anyway, due to lazy evaluation

Evaluation Strategies

- So far we have used Eval/rpar/rseq
  - these are quite low-level tools
  - but it's important to understand how the underlying mechanisms work
- Now, we will raise the level of abstraction
- Goal: encapsulate parallel idioms as re-usable components that can be composed together.

The Strategy type

```
type Strategy a = a -> Eval a
```

- A Strategy is...
  - A function that,
  - when applied to a value ‘a’,
  - evaluates ‘a’ to some degree
  - (possibly sparking evaluation of sub-components of ‘a’ in parallel),
  - and returns an equivalent ‘a’ in the Eval monad
- NB. the return value should be observably equivalent to the original
  - (why not the same? we’ll come back to that...)

Example...

```
parList :: Strategy [a]
```

- A Strategy on lists that sparks each element of the list
- This is usually not sufficient – suppose we want to evaluate the elements fully (e.g. with deep), or do parList on nested lists.
- So we parameterise parList over the Strategy to apply to the elements:

```
parList :: Strategy a -> Strategy [a]
```

Defining parList

```
type Strategy a = a -> Eval a
parList :: Strategy a -> Strategy [a]
```

- We have the building blocks:

```
rpar :: a -> Eval a
     :: Strategy a
```

```
parList :: (a -> Eval a) -> [a] -> Eval [a]
parList s []     = return []
parList s (x:xs) = do
  x'  <- rpar (runEval (s x))
  xs' <- parList s xs
  return (x':xs')
```

By why *do* Strategies return a value?

```
parList (a -> Eval a) -> [a] -> Eval [a]
parList s []     = return ()
parList s (x:xs) = do
  x'  <- rpar (runEval (s x))
  xs' <- parList s xs
  return (x':xs')
```

- Spark pool points to (runEval (s x))
- If nothing else points to this expression, the runtime will discard the spark, on the grounds that it is not required
- *Always keep hold of the return value of* rpar
- (see the notes for more details on this)

Let's generalise...

- Instead of parList which has the sparking behaviour built-in, start with a basic traversal in the Eval monad:

```
evalList :: (a -> Eval a) -> [a] -> Eval [a]
evalList f []     = return ()
evalList f (x:xs) = do
  x'  <- f x
  xs' <- parList f xs
  return (x':xs')
```

- and now:

```
parList f = evalList (rpar `dot` f)
 where s1 `dot` s2 = s1 . runEval . s2
```

Generalise further...

- In fact, evalList already exists for arbitrary data types in the form of 'traverse'.

```
evalTraversable
  :: Traversable t => Strategy a -> Strategy (t a)

evalTraversable = traverse

evalList = evalTraversable
```

- So, building Strategies for arbitrary data structures is easy, given an instance of Traversable.
- (not necessary to understand Traversable here, just be aware that many Strategies are just generic traversals in the Eval monad).

How do we *use* a Strategy?

```
type Strategy a = a -> Eval a
```

- We could just use runEval
- But this is better:

```
x `using` s = runEval (s x)
```

- e.g.

```
myList `using` parList rdeepseq
```

- Why better? Because we have a "law":
  - `x `using` s ≈ x`
  - We can insert or delete "`using` s" without changing the semantics of the program

Is that really true?

- Well, not entirely.

1. It relies on Strategies returning "the same value" (*identity-safety)*
   - Strategies from the library obey this property
   - Be careful when writing your own Strategies
2. x `using` s might do more evaluation than just x.
   - So the program with x `using` s might be _|_, but the program with just x might have a value

- if identity-safety holds, adding using cannot make the program produce a different result (other than _|_)

But we wanted ‘parMap’

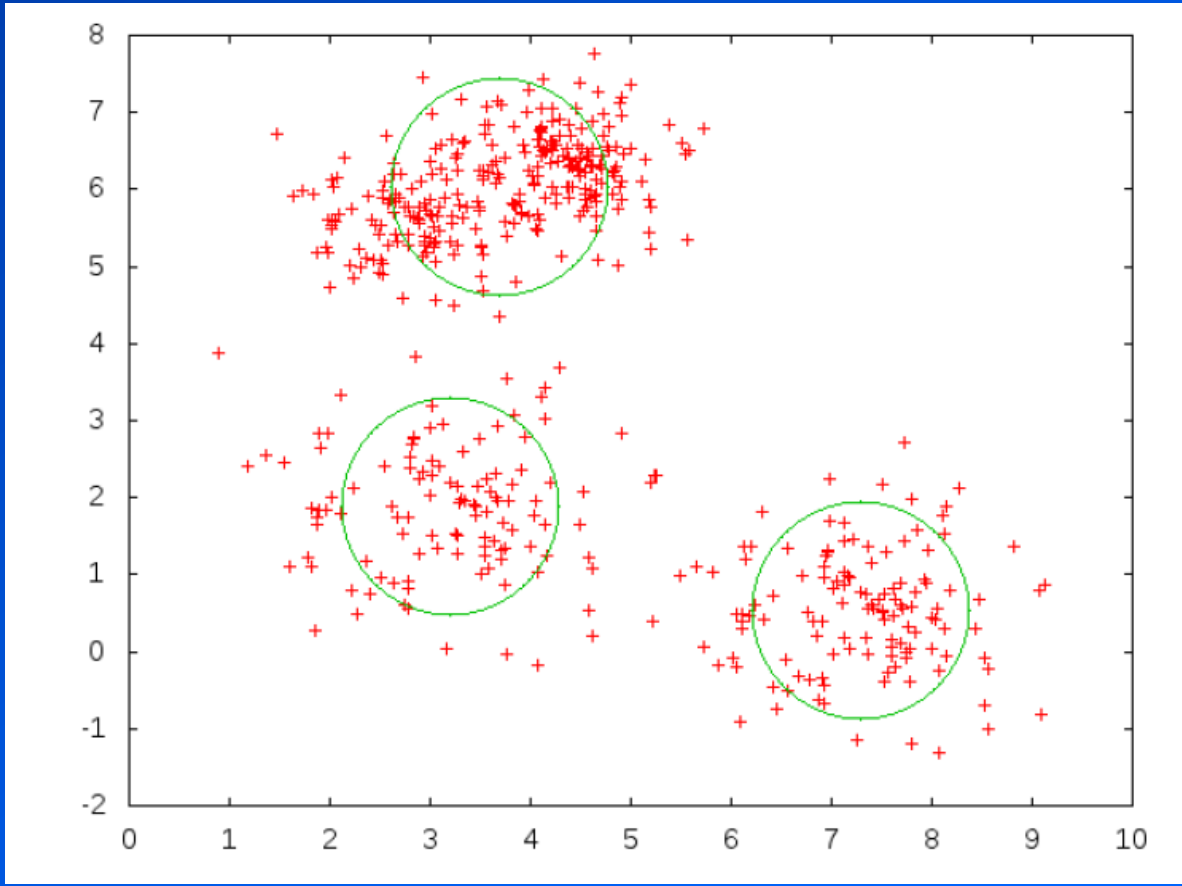
- Earlier we used parMap to parallelise Sudoku
- But parMap is a combination of two concepts:
  - The *algorithm*, ‘map’
  - The *parallelism,* ‘parList’

```
parMap f x = map f xs `using` parList
```

- With Strategies, the algorithm can be separated from the parallelism.
  - The algorithm produces a (lazy) result
  - A Strategy filters the result, but does not do any computation – it returns the same result.

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
- makeNewClusters is step 3
- 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
- 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 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

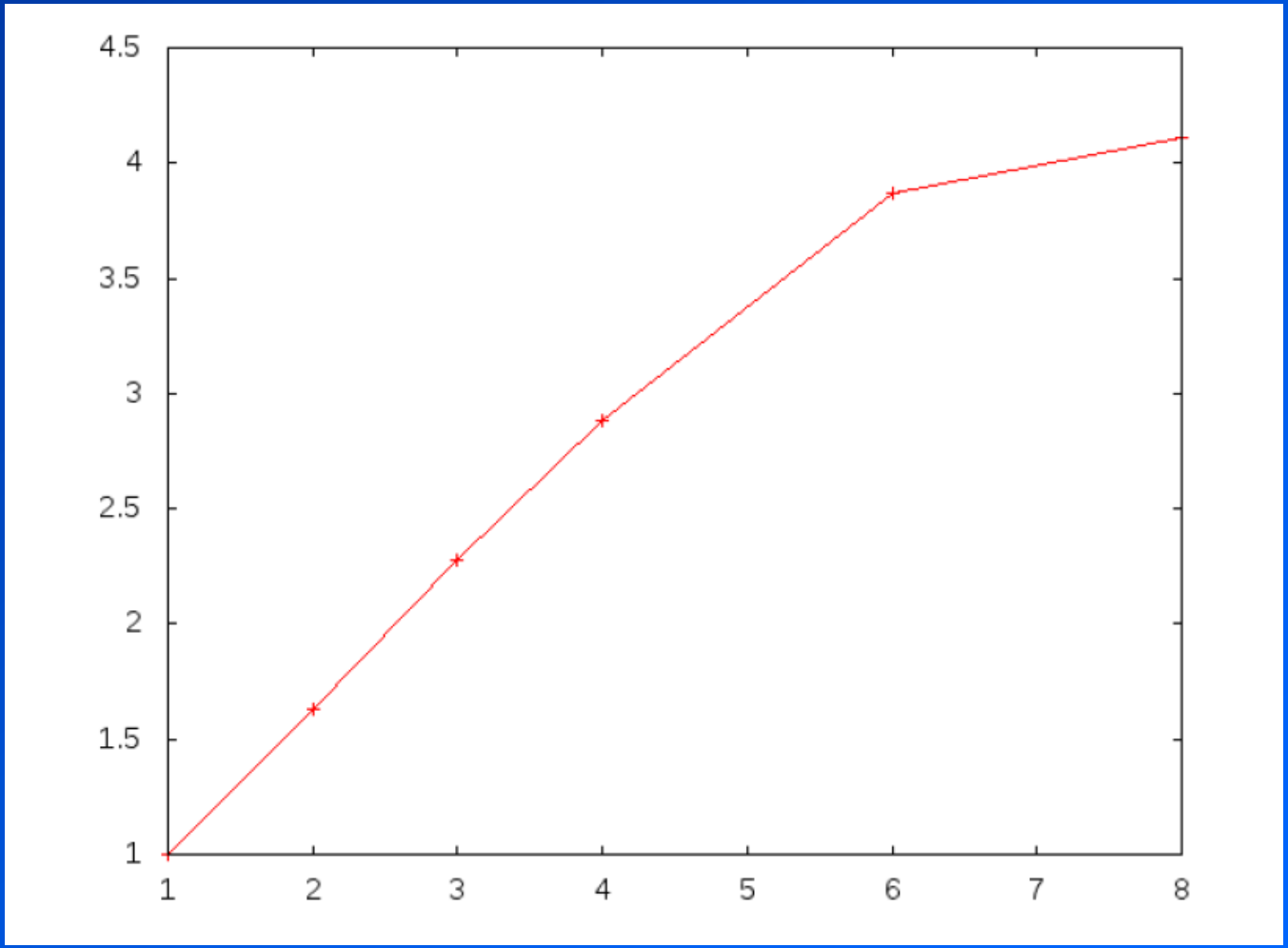
Final parallel implementation

```
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 =
                map (step nclusters clusters) chunks
                   `using` parList rdeepseq

             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

State of play

- yesterday we:
  - looked at the Eval monad, rpar and rseq, and Strategies
  - got confused about laziness
- This morning:
  - short intro to another programming model for parallelism in Haskell, the Par monad
  - Lab session (Parallel Haskell)
- This afternoon:
  - Concurrent Haskell

- Strategies, in theory:
  - *Algorithm + Strategy = Parallelism*
- Strategies, in practice (sometimes):
  - *Algorithm + Strategy = No Parallelism*
- lazy evaluation is the magic ingredient that bestows modularity, but lazy evaluation can be tricky to deal with.
- The Par monad:
  - abandon modularity via lazy evaluation
  - get a more direct programming model
  - avoid some common pitfalls
  - modularity via higher-order skeletons
  - a beautiful implementation

A menu of ways to screw up

- less than 100% utilisation
  - parallelism was not created, or was discarded
  - algorithm not fully parallelised – residual sequential computation
  - uneven work loads
  - poor scheduling
  - communication latency
- extra overhead in the parallel version
  - overheads from rpar, work-stealing, deep, ...
  - lack of locality, cache effects...
  - larger memory requirements leads to GC overhead
  - GC synchronisation
  - duplicating work

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 ()
```

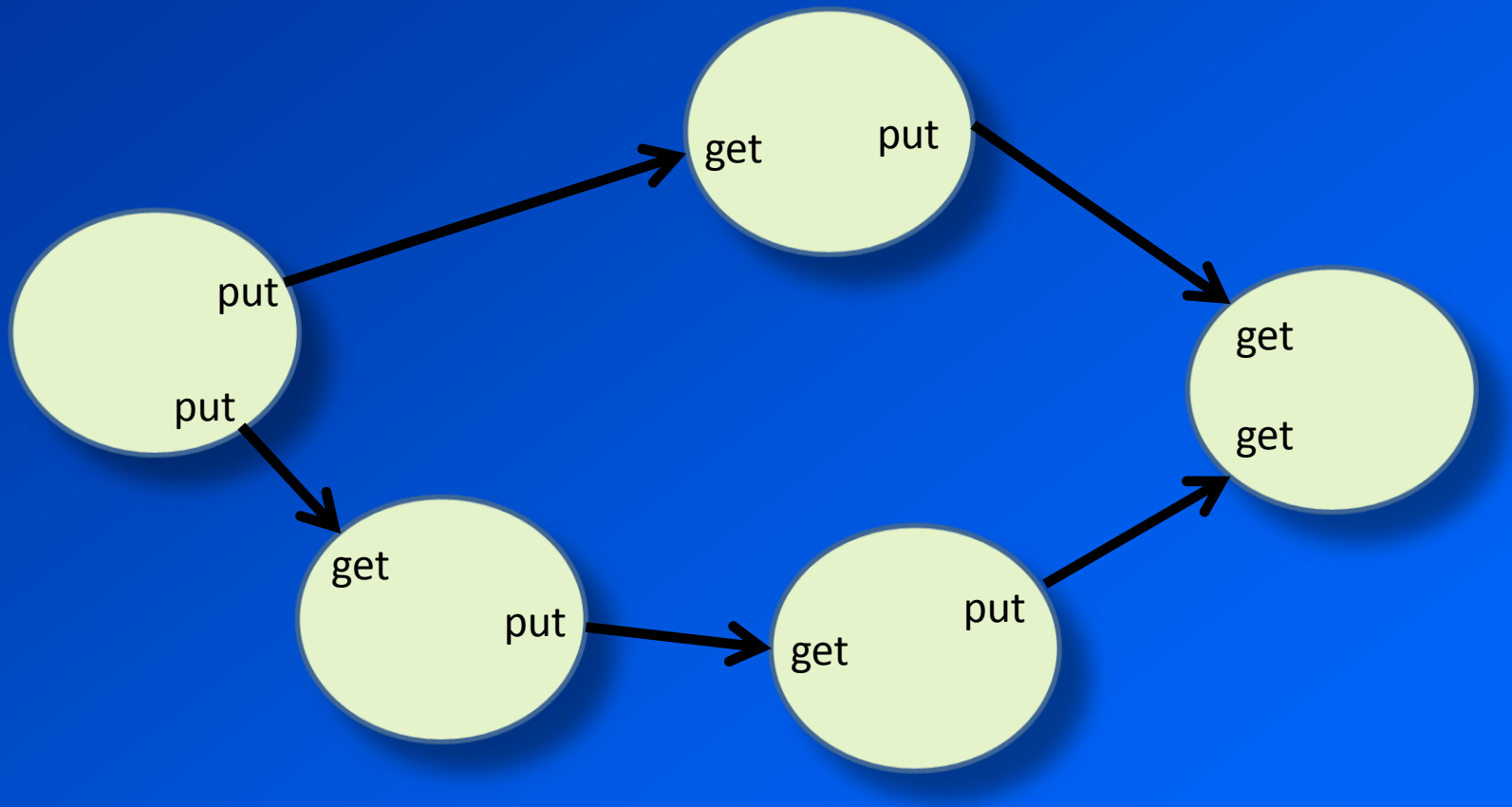
Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

forking is *explicit*

results are communicated through IVars

Par expresses dynamic dataflow



Examples

- Par can express regular parallelism, like parMap. 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 (actually parMapM):

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

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')
```

- In practice you want to use the sequential version when the grain size gets too small

How did we avoid laziness?

- put is hyperstrict.
- (by default)
- there's also a WHNF version called put_

Dataflow problems

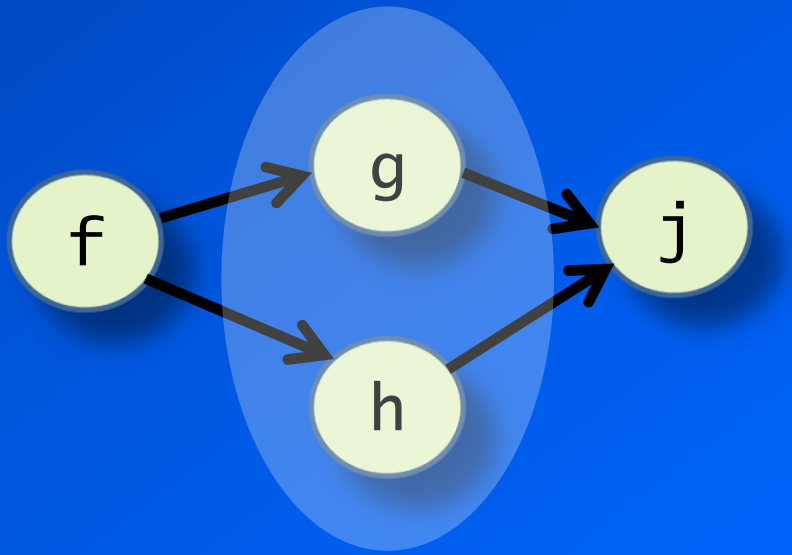
- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

Example

- Consider typechecking (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

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.

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:

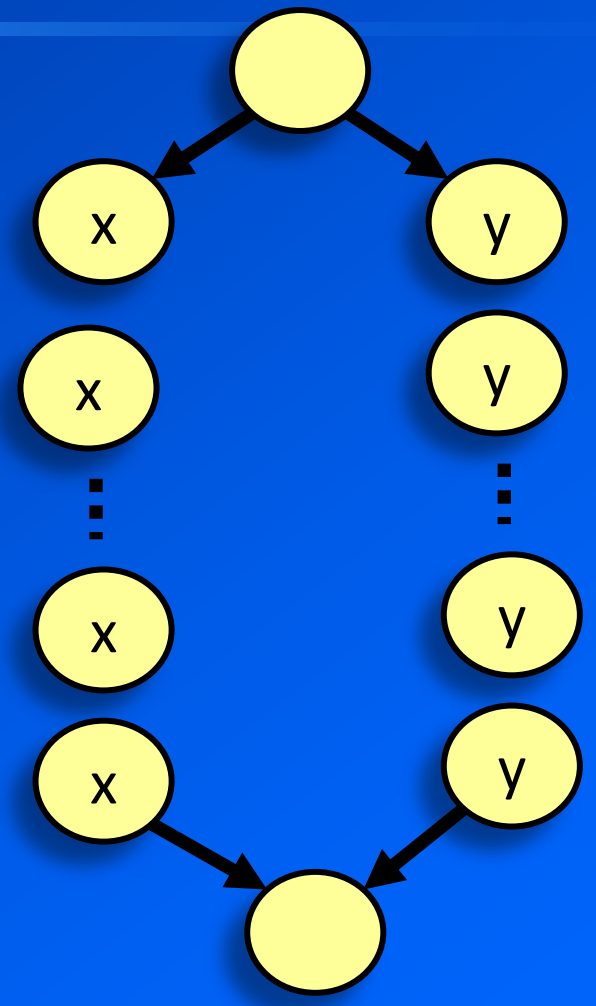
```
inferTop :: TopEnv -> Term -> Par MonoType
```

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

Thoughts to take away...

- *Parallelism is not the goal*
  - Making your program faster is the goal
  - (unlike Concurrency, which is a goal in itself)
  - If you can make your program fast enough without parallelism, all well and good
  - However, designing your code with parallelism in mind should ensure that it can ride Moore's law a bit longer
  - maps and trees, not folds

Lab

- Download the sample code here:

```
http://community.haskell.org/~simonmar/par-tutorial.tar.gz
```

- or get it with git:

```
git clone https://github.com/simonmar/par-tutorial.git
```

- code is in par-tutorial/code
- lab exercises are here:

```
http://community.haskell.org/~simonmar/lab-exercises.pdf
```

- install extra packages:

```
cabal install xml utf8-string
```

Open research problems?

- How to do safe nondeterminism
- Par monad:
  - implement and compare scheduling algorithms
  - better raw performance (integrate more deeply with the RTS)
- Strategies:
  - ways to ensure identity safety
  - generic clustering