Parallel Haskell Tutorial: The Par Monad

Simon Marlow

Ryan Newton

Simon Peyton Jones

What we're going to do…

- Parallelise some simple programs with the Par monad
- Compile and run the code, measure its performance
- Debug performance with the ThreadScope tool
- You: parallelise a sequential K-Means program

Code is here:

- Github

```
http://github.com/simonmar/par-tutorial
```

- Git

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

- Tarball:

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

- Code in par-tutorial/code
- par-tutorial/doc is “Parallel and Concurrent Programming in Haskell” (70-page tutorial)

History

- Deterministic parallel programming in Haskell
- par/pseq (1996)
  - Simple, elegant primitives
  - Pure, deterministic parallelism for a lazy programming language
- Evaluation Strategies (1998, revised 2010)
  - Added parallelism over data structures
    - (for example, evaluating the elements of a list in parallel)
  - Composability
  - Modularity

Not a bed of roses…

- Strategies, in theory:
  - *Algorithm + Strategy = Parallelism*
- Strategies, in practice (sometimes):
  - *Algorithm + Strategy = No Parallelism*
- laziness is the magic ingredient that bestows modularity, but laziness can be tricky to deal with.
- Beyond the simple cases, Strategies requires the programmer to understand subtle aspects of the execution model (laziness, garbage collection)
- Diagnosing performance problems is really hard

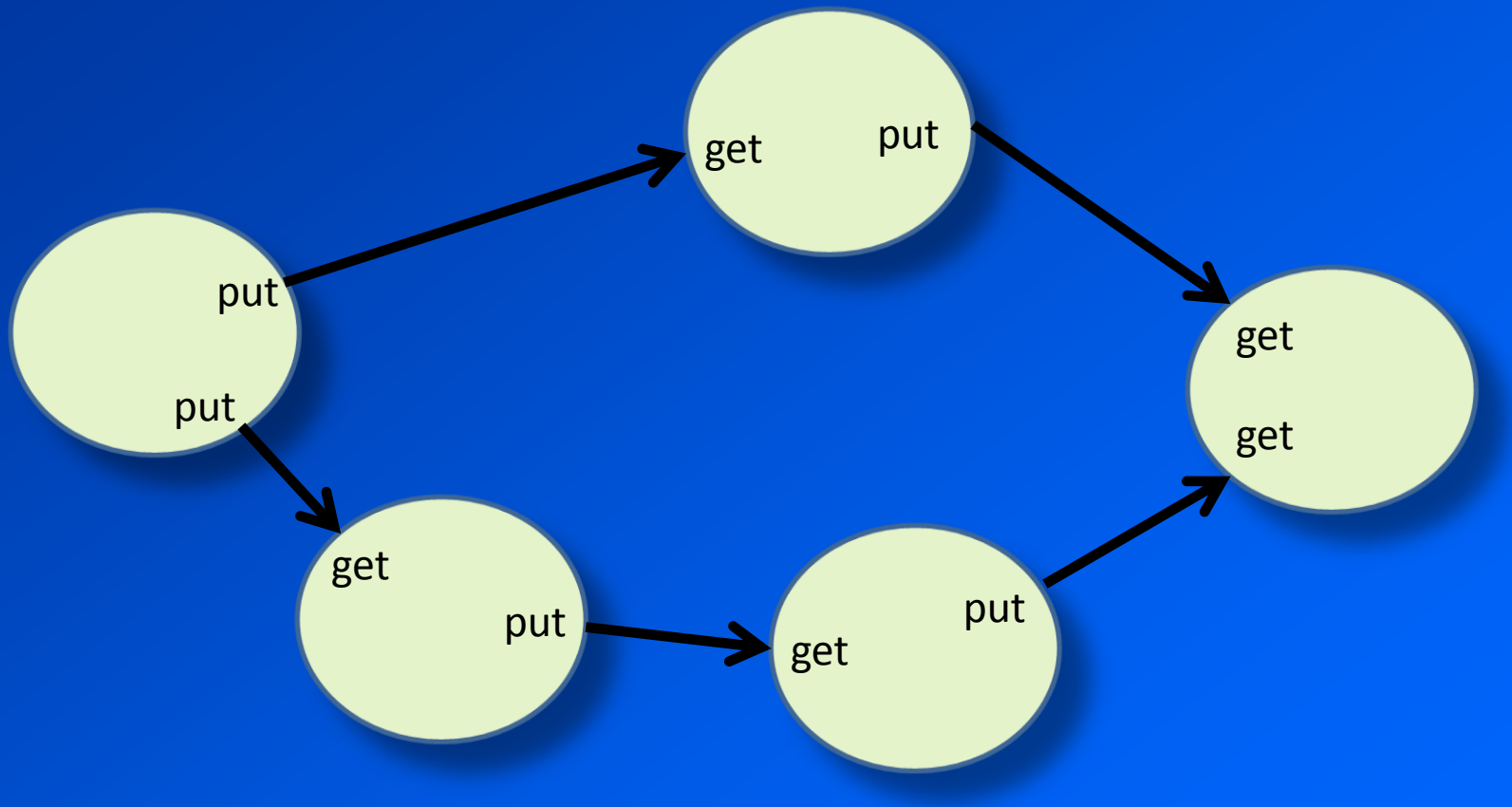
The Par Monad

- 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 magic here, just standard Haskell abstraction techniques)
  - 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

```
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

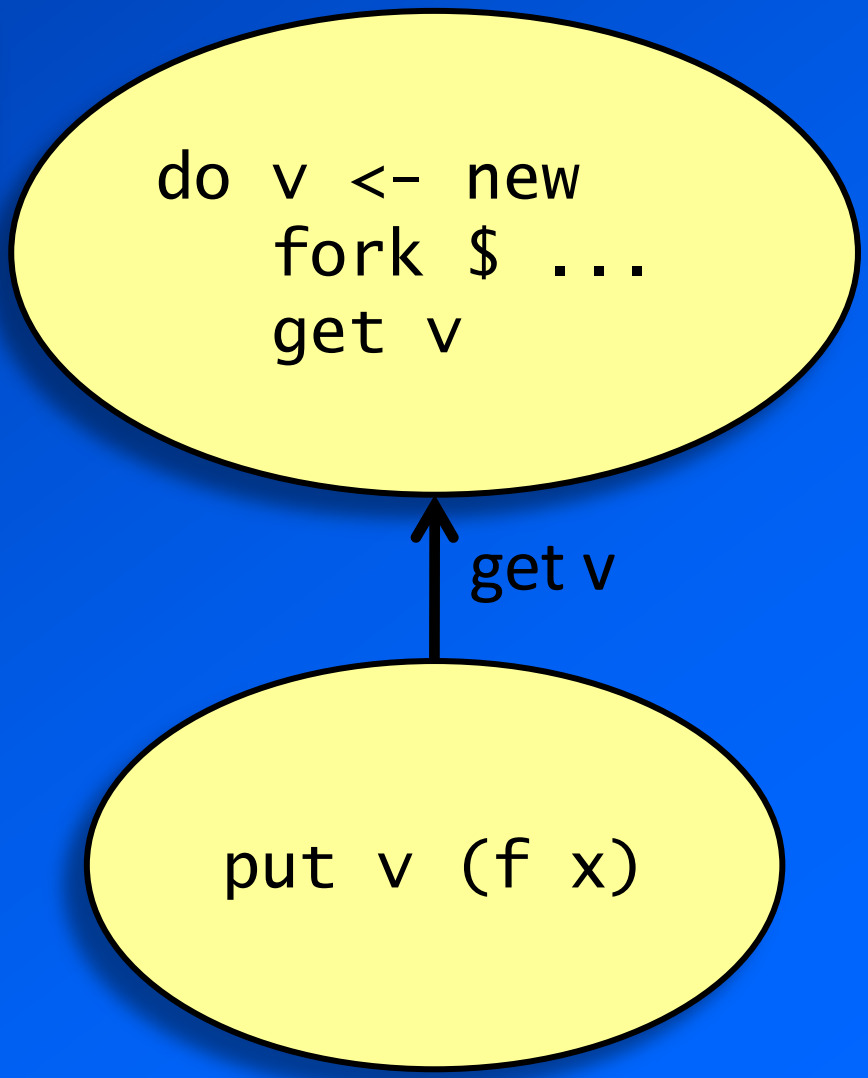
- *put is fully strict*

```
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

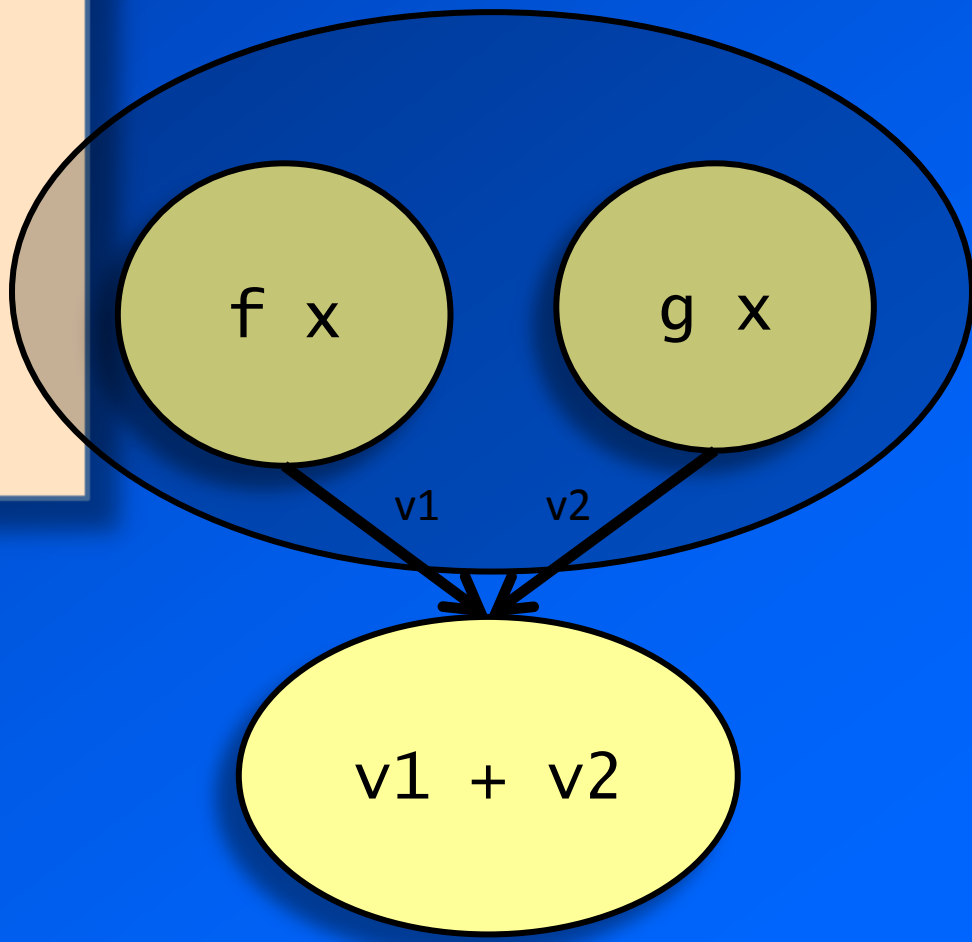
How does this make a dataflow graph?

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



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)
```



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
import Data.Maybe

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

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

Compile the program...

```
$ ghc --make -O2 sudoku-par1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku-par1.hs, sudoku-par1.o )
Linking sudoku-par1 ...
$
```

Run the program...

```
$ ./sudoku-par1 sudoku17.1000.txt +RTS -s
./sudoku-par1 sudoku17.1000.txt +RTS -s
1000
   2,392,198,136 bytes allocated in the heap
      38,689,840 bytes copied during GC
         213,496 bytes maximum residency (14 sample(s))
          94,480 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

…

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.88s  (  2.88s elapsed)
  GC    time    0.14s  (  0.14s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.02s  (  3.02s elapsed)

…
```

Now to parallelise it

- I have two cores on this laptop, so why not divide the work in two and do half on each core?

Sudoku solver, version 2

```
import Sudoku
import Control.Exception
import System.Environment
import Data.Maybe
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 one processor first

```
> ./sudoku-par2 sudoku17.1000.txt +RTS -s
./sudoku-par2 sudoku17.1000.txt +RTS -s
1000
   2,400,398,952 bytes allocated in the heap
      48,900,472 bytes copied during GC
       3,280,616 bytes maximum residency (7 sample(s))
         379,624 bytes maximum slop
              11 MB total memory in use (0 MB lost due to fragmentation)

  …

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.91s  (  2.91s elapsed)
  GC    time    0.19s  (  0.19s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.09s  (  3.09s elapsed)
 …
```

A little slower (was 3.02 before). Splitting and reconstructing the list has some overhead.

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)
…
```

-N2 says “use 2 OS threads”
Only available when the program was compiled with -threaded

Speedup, yay!

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 sudoku-par2: 3.02/1.99 = 1.52
  - not great...

Why not 2?

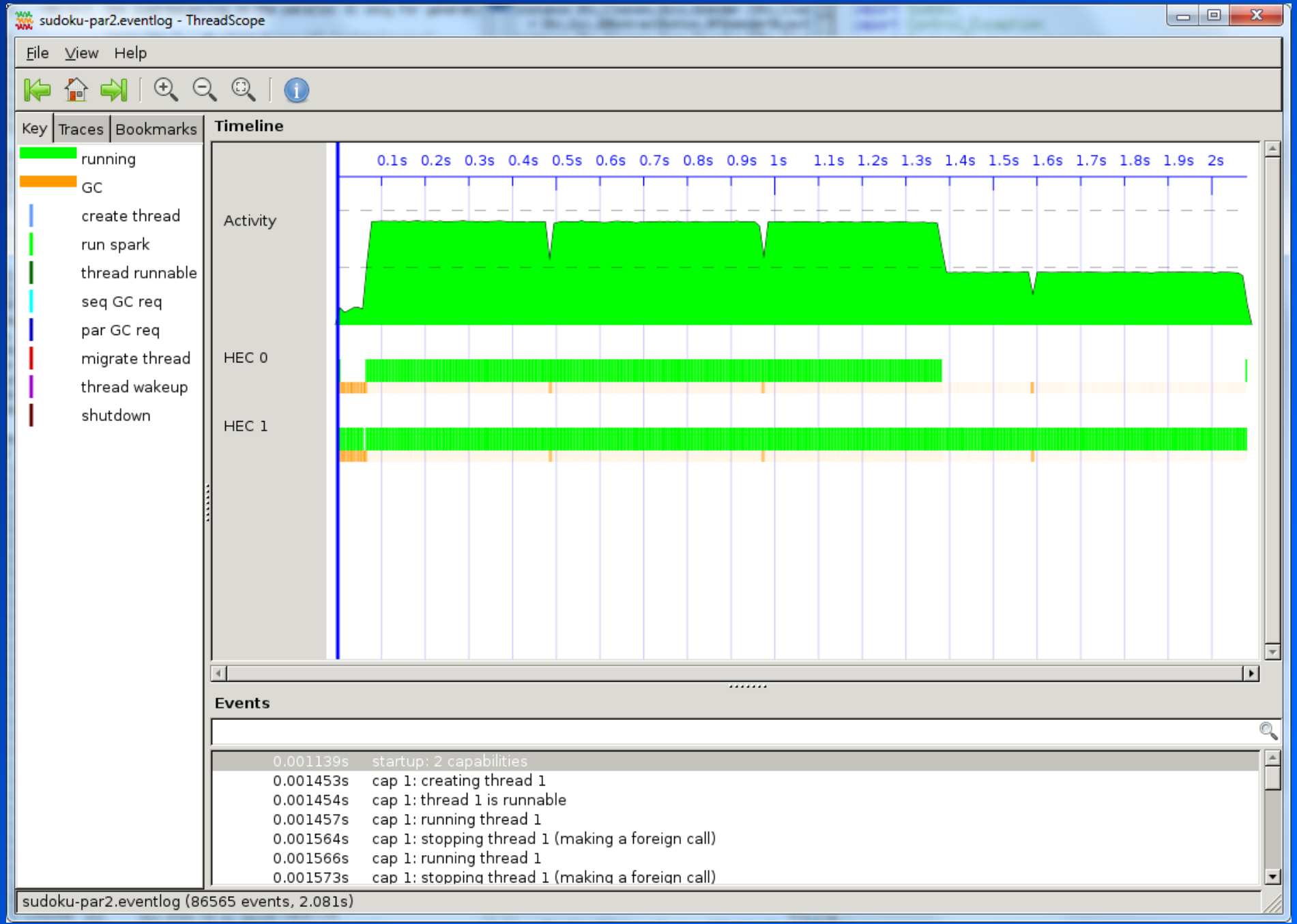
- there are two reasons for lack of parallel speedup:
  - less than 100% utilisation (some processors idle for part of the time)
  - extra overhead in the parallel version
- Each of these has many possible causes...

A menu of ways to get poor speedup

- less than 100% utilisation
  - Not enough parallelism in the algorithm, or uneven work loads
  - poor scheduling
  - communication latency
- extra overhead in the parallel version
  - Algorithmic overheads
  - larger memory requirements leads to GC overhead
  - lack of locality, cache effects...
  - GC synchronisation
  - duplicating work

So how do we find out what went wrong?

- We need profiling tools.
- GHC has ThreadScope
- Use ThreadScope like this:
  - Add “-eventlog” when compiling
  - Add “+RTS -ls” when running (“l” for “log”, “s” for “scheduler events”).
  - Program generates <prog>.eventlog
  - Run “threadscope <prog>.eventlog”


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)

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*
- The Par monad has a scheduler built-in
  - So all we have to do is break the work into smaller units
- Benefits:
  - copes with problems that have unknown or varying distributions of work
  - adapts to different number of processors: the same program scales over a wide range of cores
- Notes
  - Unlike sparks, Par can handle an unlimited number of tasks (until memory runs out)
  - fork in the Par monad is much cheaper than forkIO (approx 10x), but not as cheap as rpar.

parMap

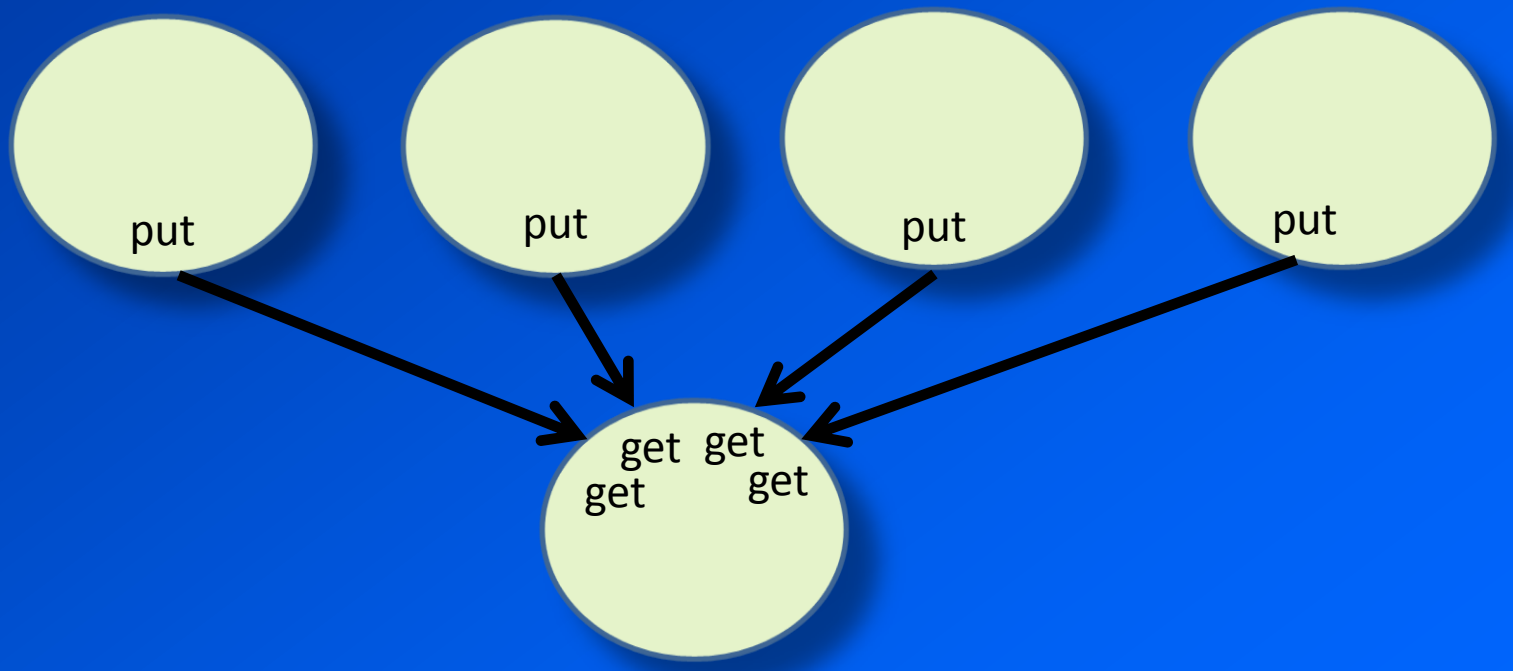
- parMap, as its name suggests, is a parallel *map*. First expand our vocabulary a bit:

```
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

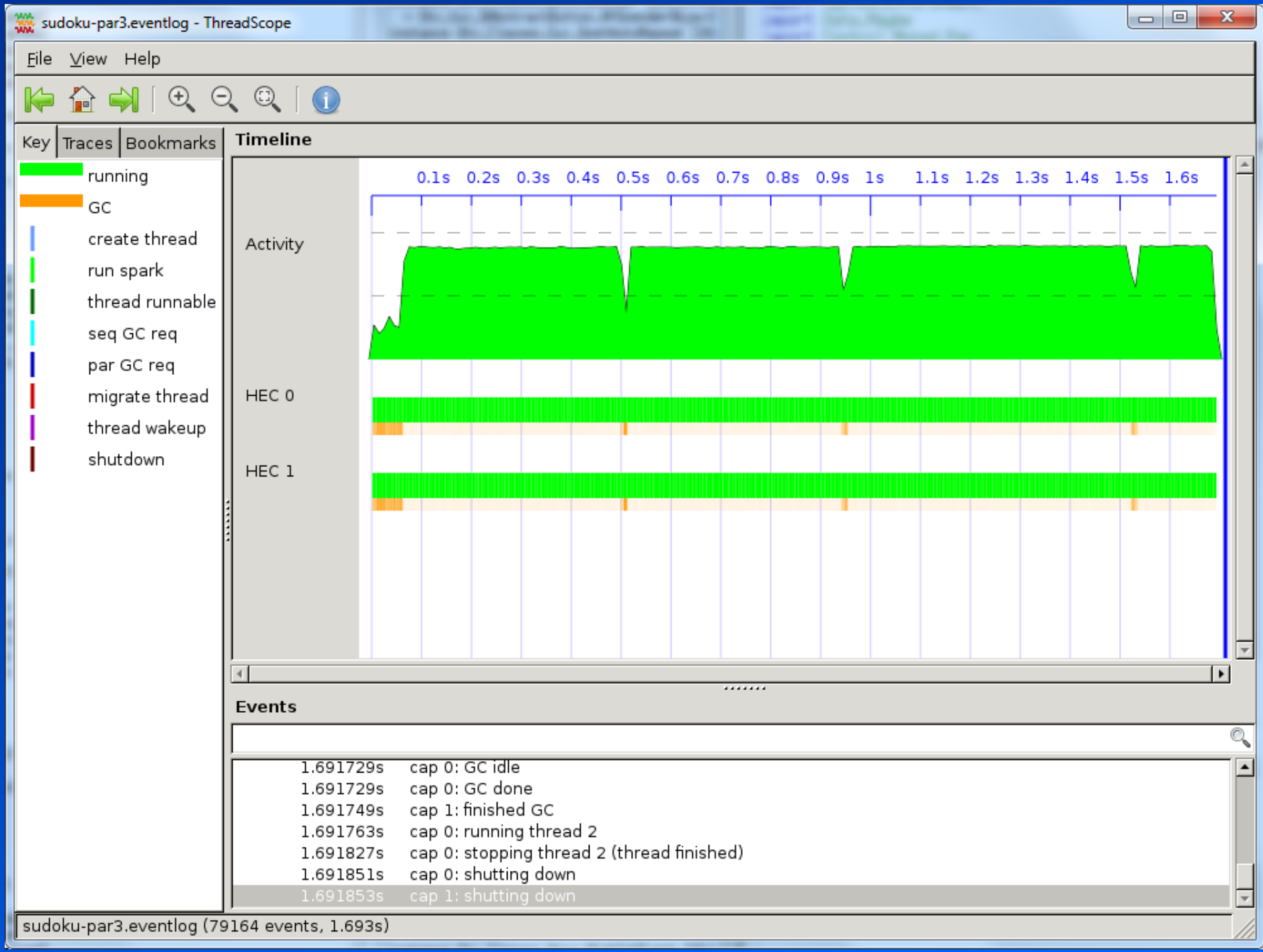
```
import Sudoku
import Control.Exception
import System.Environment
import Data.Maybe
import Control.Monad.Par

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

- far simpler than the version that split the input into two lists
- How does it perform?

sudoku-par3 on 2 cores

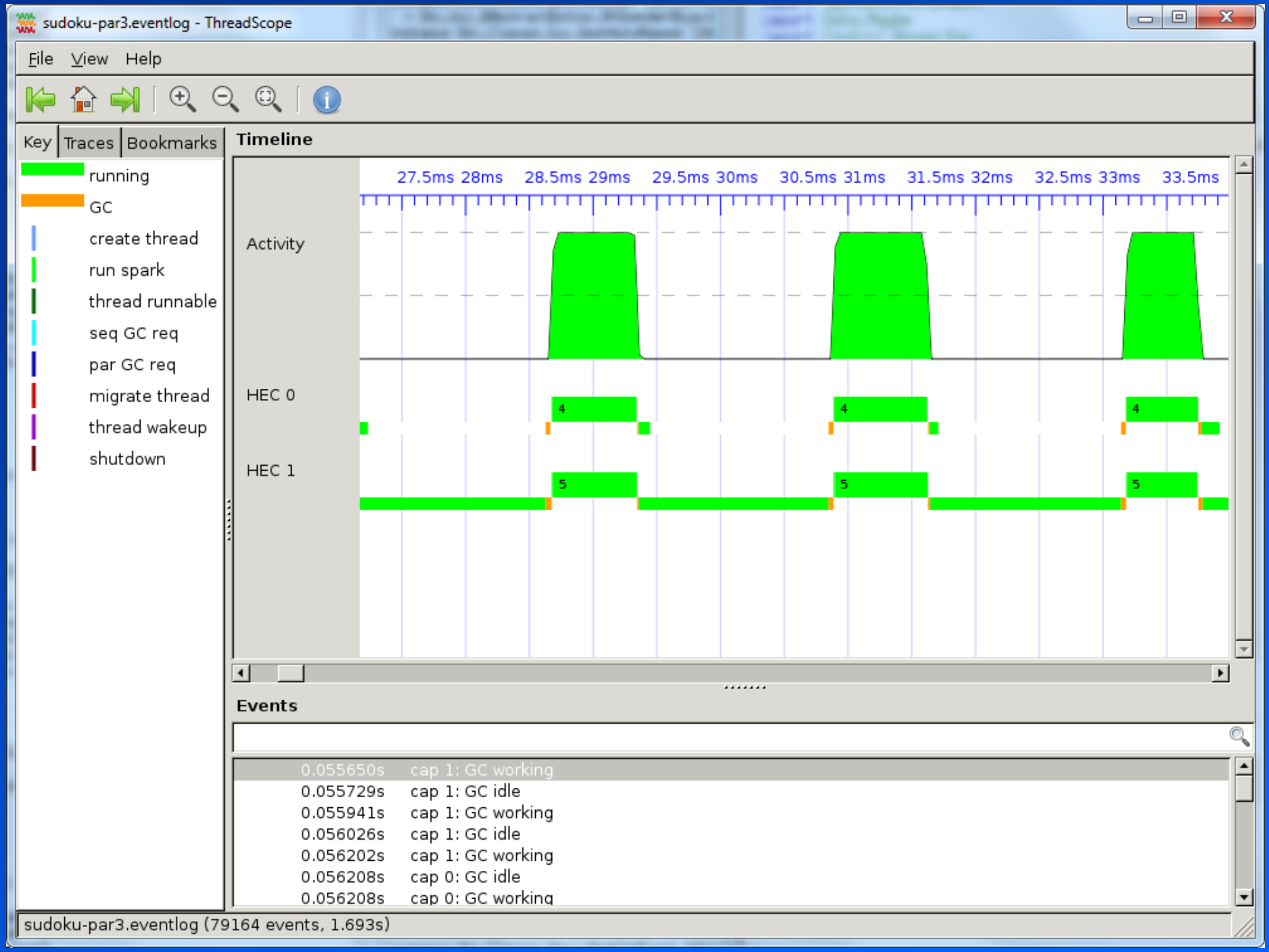
```
./sudoku-par3 sudoku17.1000.txt +RTS -N2 -s
1000
   2,400,973,624 bytes allocated in the heap
      50,751,248 bytes copied during GC
       2,654,008 bytes maximum residency (6 sample(s))
         368,256 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)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

  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)

Why only 1.79?

- That bit at the start of the profile doesn’t look right, let’s zoom in…


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
27.5ms 28ms 28.5ms 29ms 29.5ms 30ms 30.5ms 31ms 31.5ms 32ms 32.5ms 33ms 33.5ms
Activity
HEC 0
HEC 1
4
4
4
5
5
5
Events
0.055650s cap 1: GC working
0.055729s cap 1: GC idle
0.055941s cap 1: GC working
0.056026s cap 1: GC idle
0.056202s cap 1: GC working
0.056208s cap 0: GC idle
0.056208s cap 0: GC working
sudoku-par3.eventlog (79164 events, 1.693s)

Why only 1.79?

- It looks like these garbage collections aren't very parallel (one thread is doing all the work)
- Probably: lots of data is being created on one core
- Suspect this is the parMap forcing the list of lines from the file (lines is lazy)
- Note in a strict language you would have to split the file into lines first
  - in Haskell we get to overlap that with the parallel computation

A note about granularity

- Granularity = size of the tasks
  - Too small, and the overhead of fork becomes significant
  - Too large, and we risk underutilisation (see sudoku-par2.hs)
  - The range of “just right” tends to be quite large
- Let’s test that. How do we change the granularity?

parMapChunk

```
parMapChunk :: NFData b => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs = fmap concat $ parMap (map f) (chunk n xs)

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:
  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 add more parallelism
- Around 1000 tasks is typically good for <16 cores

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')
```

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

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

But we can still abstract

- Higher order skeletons

```
divConq :: NFData sol
        => (prob -> Bool)   -- indivisible?
        -> (prob -> [prob]) -- split into subproblems
        -> ([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
         sols <- parMapM go (split prob)
         return (join sols)
```

Rule of thumb

- Try to separate the *application code* from the *parallel coordination* by using higher-order skeletons
  - parMap is a simple example of a skeleton
- Note:
  - not like Strategies where we separate the application code from the parallelism using *lazy data structures*
  - Not always convenient to build a lazy data structure
  - Skeletons are just higher-order programming, all communication is strict

Dataflow problems

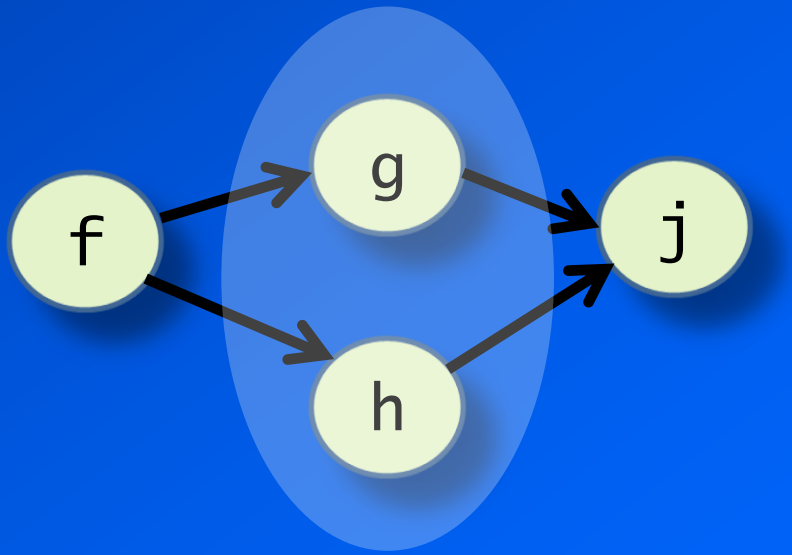
- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

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.

```
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
```

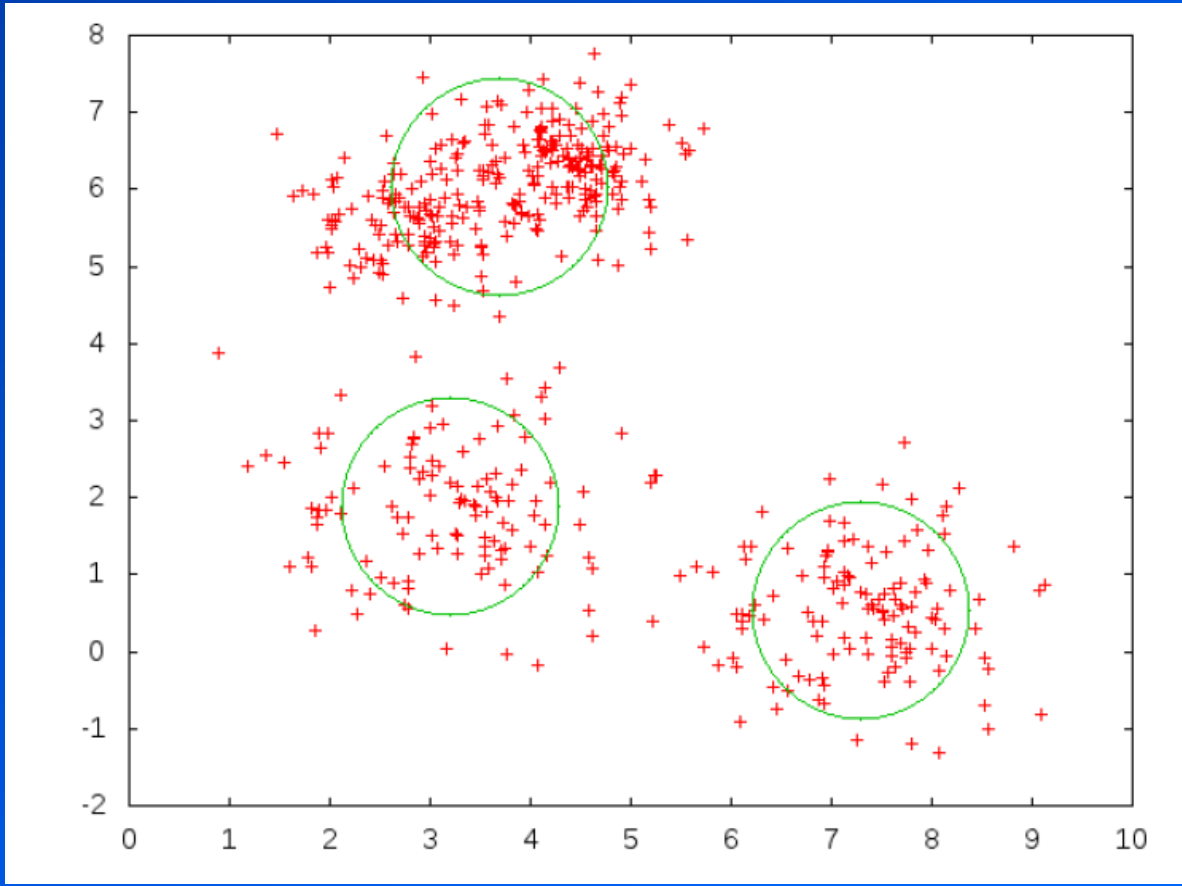
Results

```
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

Exercise: 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!

Setup

- Basic sequential algorithm: `code/kmeans/kmeans-seq.hs`
  - uses subsidiary module KMeansCommon.hs, reads data files “points.bin” and “clusters”
  - A Par-monad solution is in kmeans.hs
  - Other solutions (using Strategies): kmeans.hs, kmeans2.hs, kmeans3.hs
  - Discussion of the Strategies solution (Par monad is basically the same) in “Parallel and Concurrent Programming in Haskell”
  - NB. Use the runtime reported by the program, not +RTS –s (the program spends a lot of time reading the data files when it starts up)
  - I get 3.1 speedup on 4 cores; can you do better?

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
    - turn folds of associative operations into map/reduce

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

Resources

- These slides:

http://community.haskell.org/~simonmar/CUFP2011.pdf

- Code samples and exercise (subdirectory code/)

http://github.com/simonmar/par-tutorial
git clone http://github.com/simonmar/par-tutorial.git
http://community.haskell.org/~simonmar/par-tutorial.tar.gz

- Tutorial paper: “Parallel and Concurrent Programming in Haskell”
  - has a short section on the Par monad

http://community.haskell.org/~simonmar/par-tutorial.pdf

  - Paper “A Monad for Deterministic Parallelism” (Marlow, Newton, Peyton Jones), Haskell Symposium 2011

http://community.haskell.org/~simonmar/papers/monad-par.pdf