Parallel Haskell with the Par Monad

Simon Marlow
Facebook


Norwegian Sea
Sweden
Norway
Finland
Saint Petersburg
Санкт-Петербург
Oslo
Stockholm
Helsinki
Tallinn
Estonia
Baltic Sea
Göteborg
Latvia
North Sea
Moscow
Москва
Denmark
Lithuania

Parallelism is everywhere

- Hardware
  - Many computers
  - Multiple cores / hardware threads
  - GPUs
  - …
- Software
  - Many services
  - Many clients of a service
  - Many shards of a database
  - …

Parallel

- multi-core programming
- get the answer quicker
- deterministic (usually)
- declarative

- image manipulation
- machine learning
- database join
- spreadsheet calculation

Concurrent

- multi-threaded programming
- do things at the same time
- nondeterministic
- imperative (usually)

- web server
- GUI
- chat server
- telephone exchange

Different tradeoffs need different APIs

Landscape

- Parallel
  - par/pseq
  - Strategies
  - Par Monad
  - Repa
  - Accelerate
  - DPH
- Concurrent
  - forkIO
  - MVar
  - STM
  - async
  - Cloud Haskell

Haxl?

Parallel FP at Facebook

```
friendsOf x `intersect` friendsOf y
```

- Calculate the set of friends in common between users x and y
- friendsOf is a remote database query
- Must perform two friendsOf calls in parallel
- Our solution: an Applicative/Monad with implicit concurrency (or parallelism?)

What we're going to do today…

- Parallelise some simple programs with the Par monad
- Compile and run the code, measure its performance.
  - Learn about measuring speedup
- Debug performance with the ThreadScope tool
- Look at plenty of examples.

History

- 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

Use lazy evaluation for parallelism?

- Strategies is based on lazy evaluation:
  - computation builds a lazy data structure
  - evaluate it in parallel
- Separates computation from parallelism
  - *modularity*
- But
  - dependencies are implicit
  - understanding evaluation order can be tricky

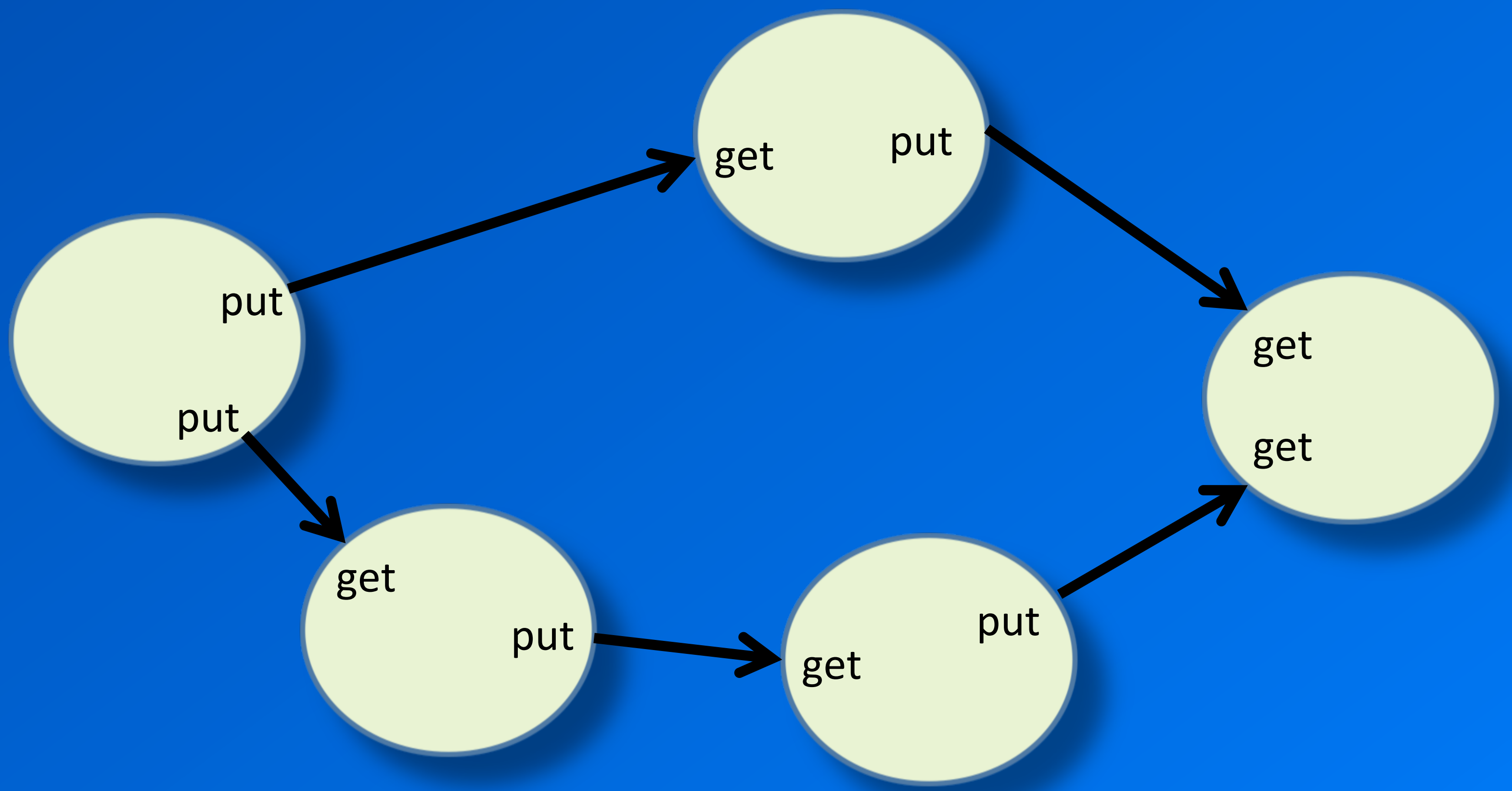
The Par Monad

- Doesn't rely on lazy evaluation
- Dependencies are explicit
- 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 a
instance Monad Par

runPar :: Par a -> a

fork :: Par () -> Par ()

data IVar a
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)

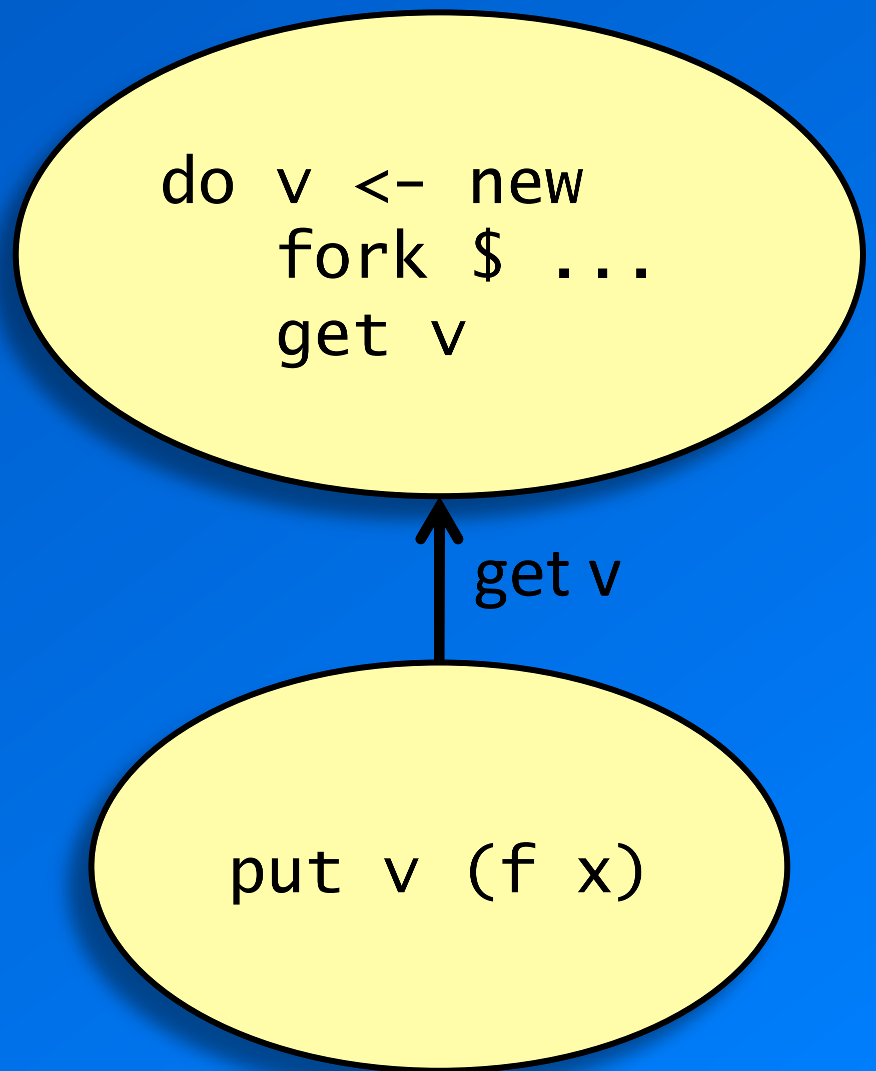
forking is *explicit*

results are communicated through IVars

How does this make a dataflow graph?

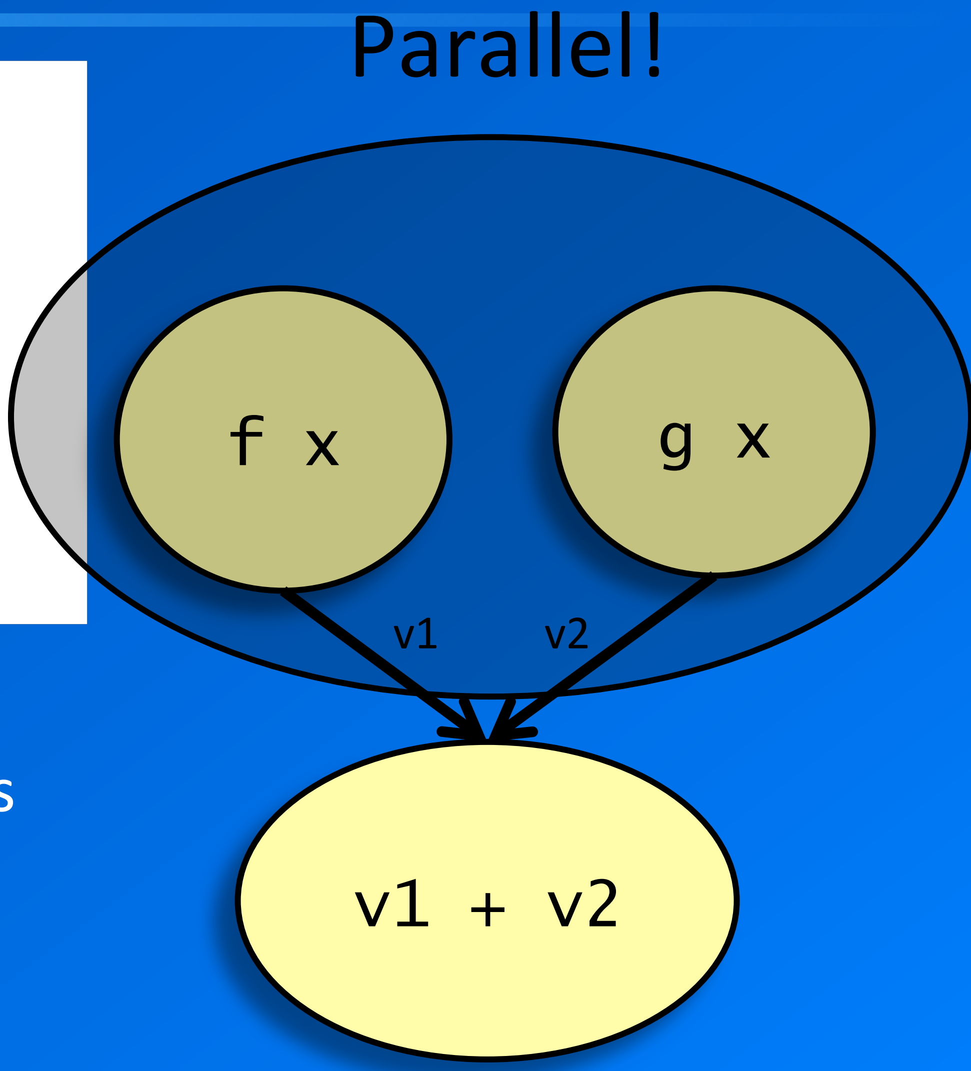
```
do v <- new
   fork $ put v (f x)
   get v
```

- 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 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

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...........
```

```haskell
import Sudoku

solve :: String -> Maybe Grid
```

Solving Sudoku problems

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

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

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

Compile the program...

```
$ ghc -O2 sudoku-par1.hs
[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

- Divide the work in two:

```haskell
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 -O2 sudoku-par2.hs -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...

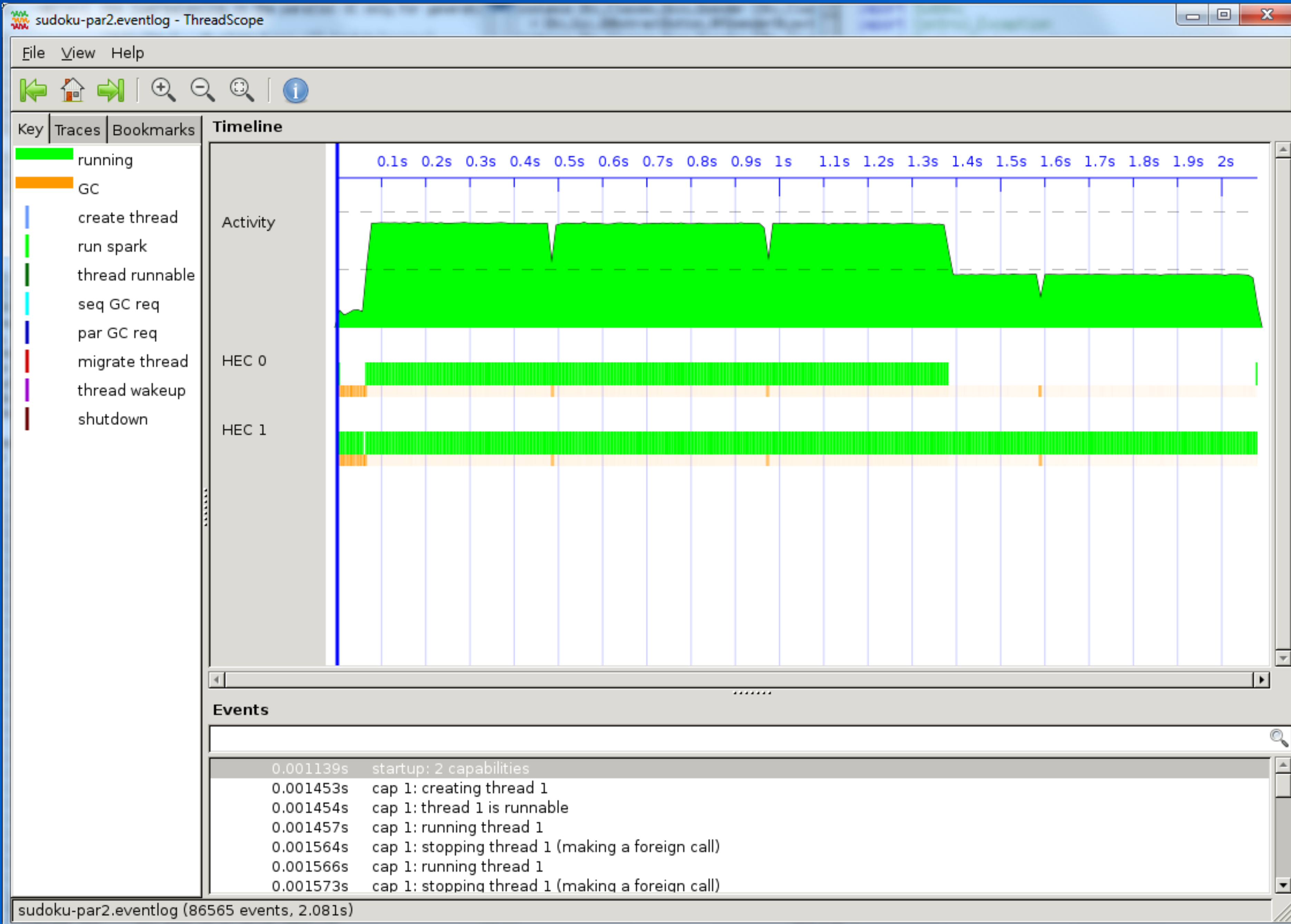
A menu of ways to get poor speedup

- less than 100% utilisation
  - Not enough parallelism in the algorithm
  - Uneven work loads
- Extra overhead due to parallelism
  - Algorithmic overheads
  - Larger memory requirements leads to GC overhead
- Other overheads in the runtime

So how do we find out what went wrong?

- We need profiling tools.
- GHC has *ThreadScope*
- Use *ThreadScope* like this:

```
$ ghc -O2 sudoku-par2.hs -threaded -eventlog
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku-par2.hs, sudoku-par2.o )
Linking sudoku-par2 ...
$ ./sudoku-par2 +RTS -N2 -l
$ threadscope sudoku-par2.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 it depends on how quickly the search finds the solution

Partitioning

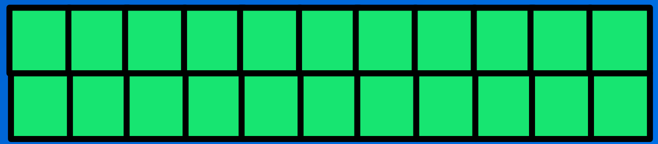
```
let (as,bs) = splitAt (length grids `div` 2) grids
```

- Dividing up the work into a small, fixed number of chunks is often bad.
  - leads to underutilisation if the chunks are uneven
  - limits the amount of parallelism (2 here)

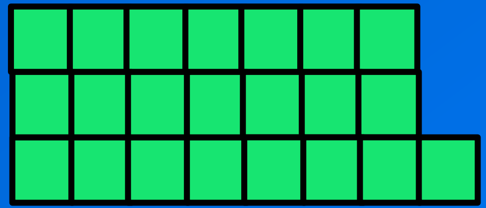
Partitioning

- The Par monad has a scheduler built-in
  - it spreads the work across the available processors
- We just need to create enough work by calling fork more often, with smaller work items.
  - the Par monad scheduler can then make better use of our processors.

Large work items



Small work items



Adding a processor

parMap

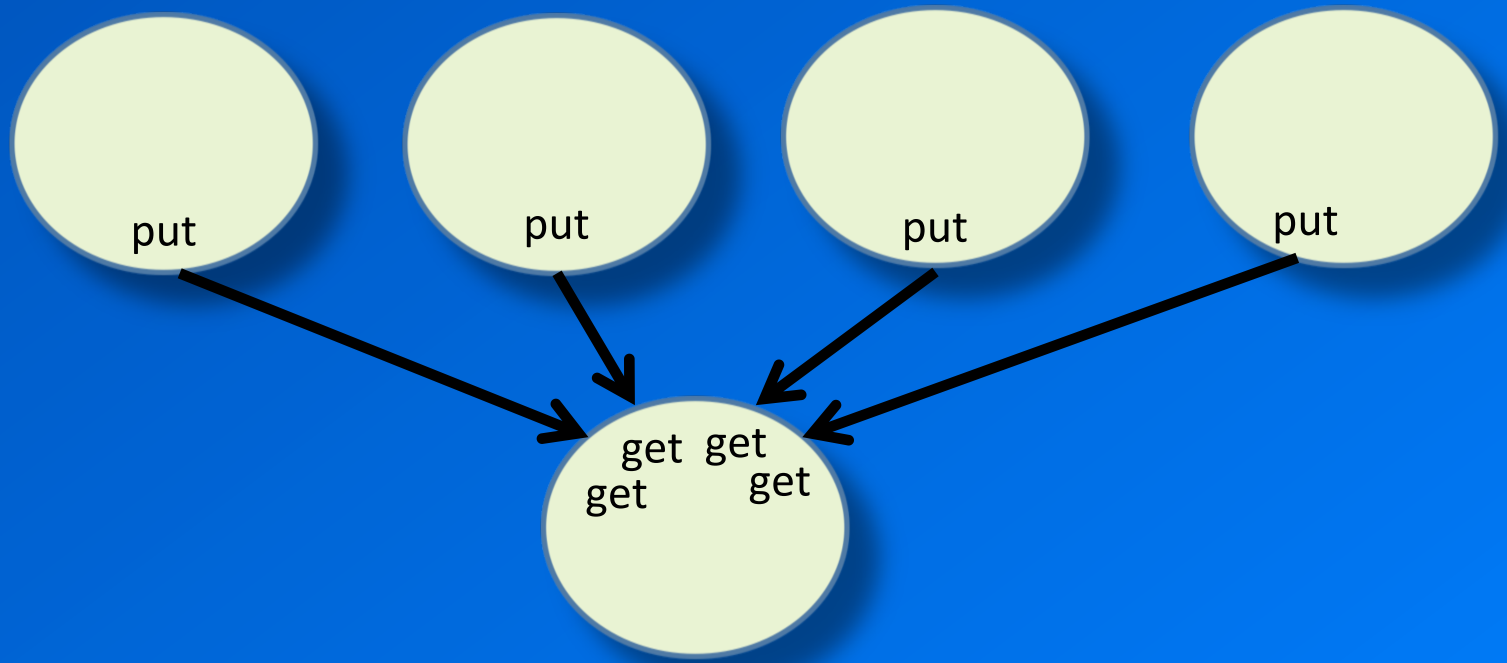
- Let’s use the Par monad to define the parMap pattern. First expand our vocabulary a bit:

```
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

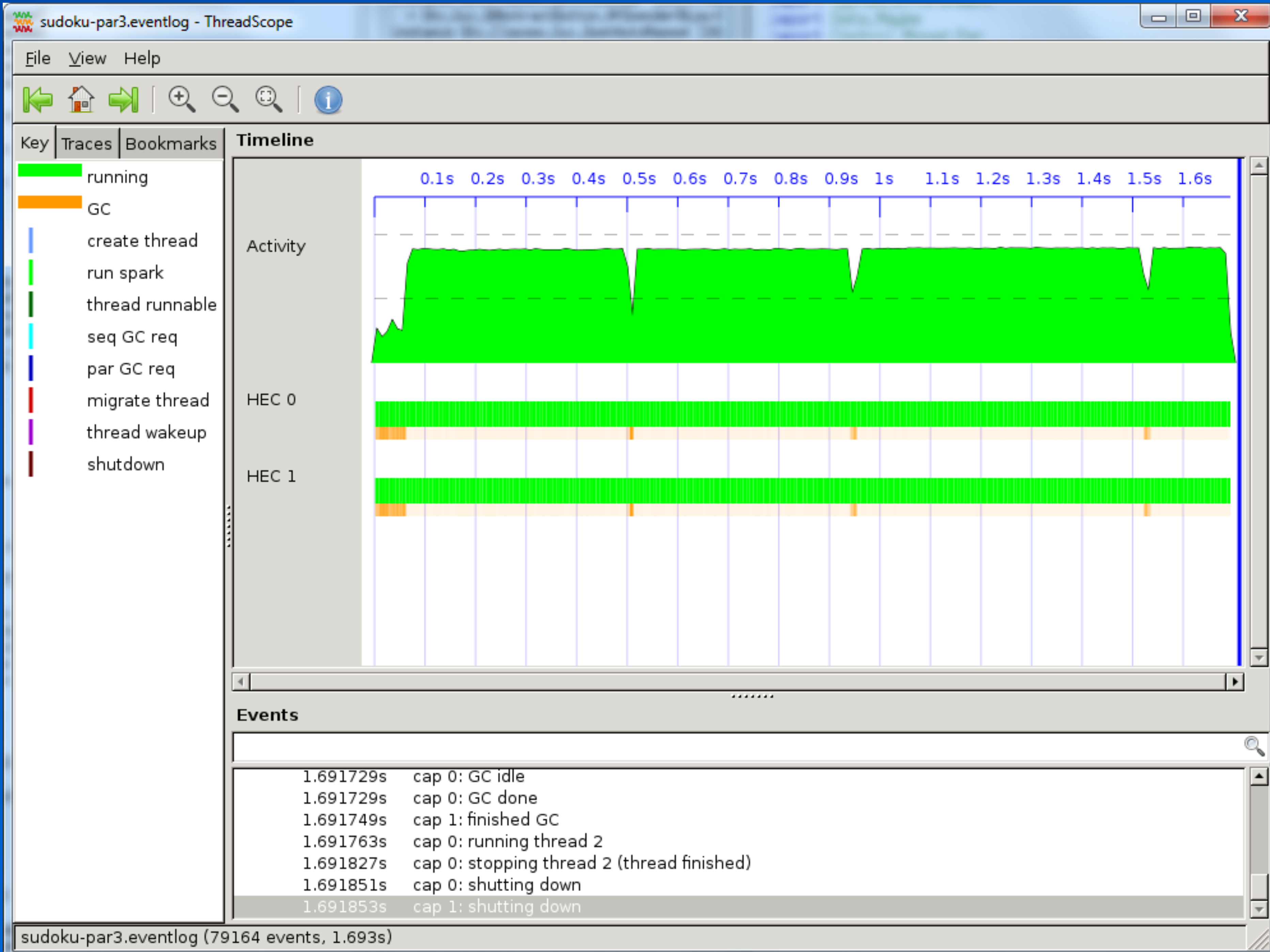
```
main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    print $ length $ filter isJust $ runPar $ parMap solve grids
```

- Much simpler than splitting into two lists
- How does it perform?

sudoku-par3 on 2 cores

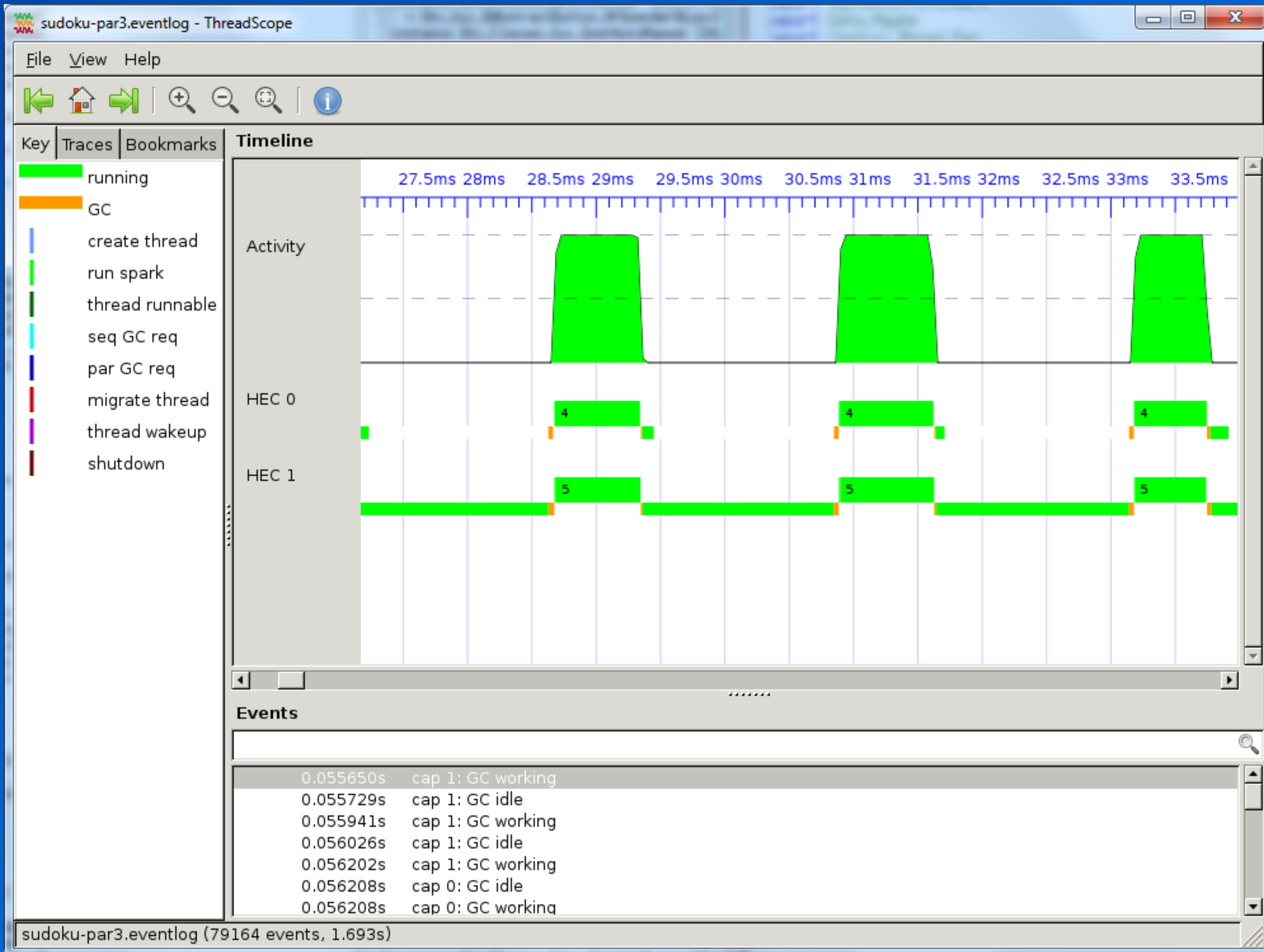
```
./sudoku-par3 sudoku17.1000.txt +RTS -N2 -s
1000
   2,400,973,624 bytes allocated in the heap
      50,751,248 bytes copied during GC
       2,654,008 bytes maximum residency (6 sample(s))
         368,256 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)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

  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
Timeline
running
GC
create thread
run spark
thread runnable
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
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
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

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 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’)
```

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

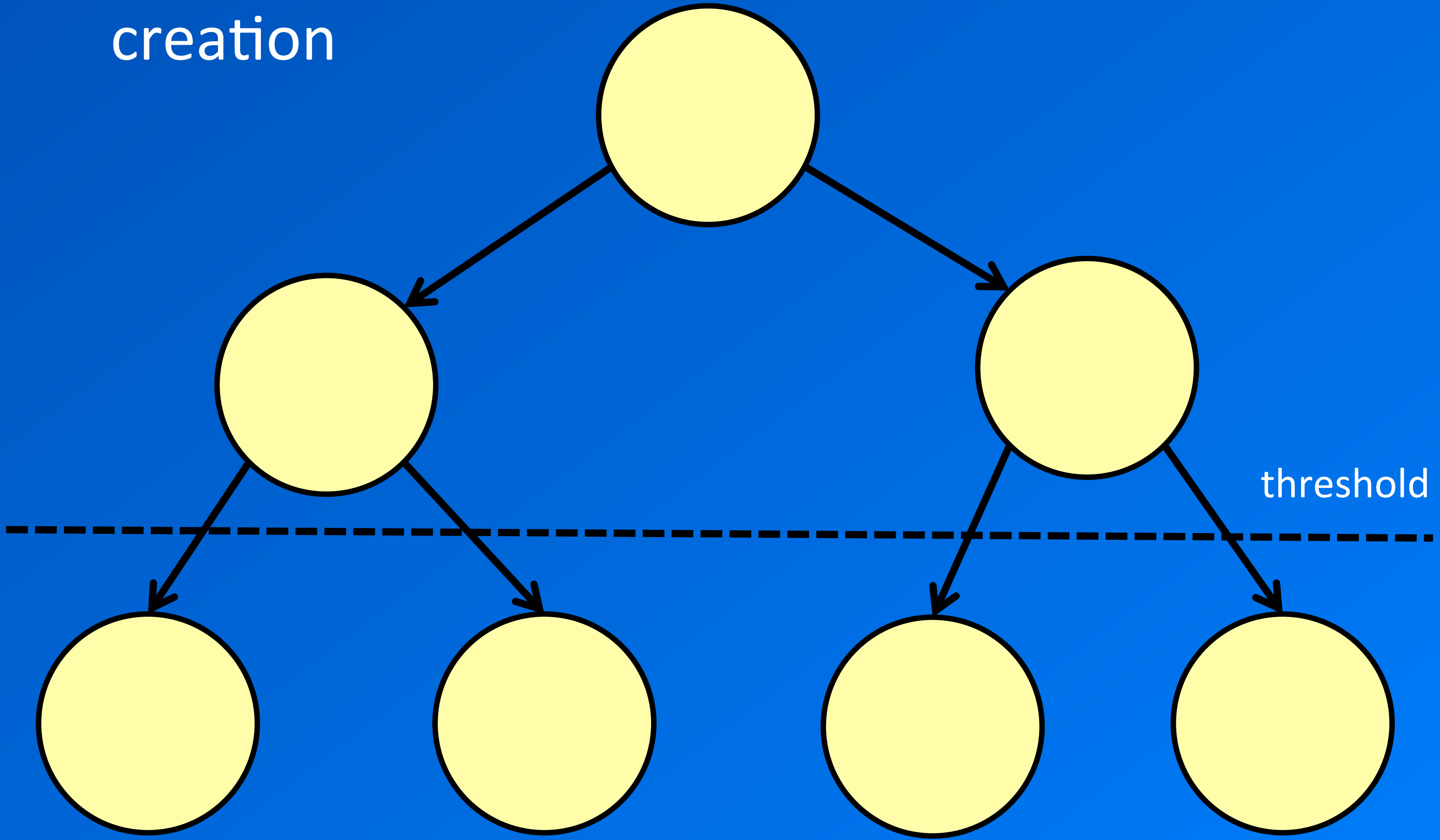
Granularity in divide-and-conquer

- 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 *depth 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)
     y <- spawn $ parfib (n-2)
     x’ <- get x
     y’ <- get y
     return (x’ + y’)

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

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

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

Dataflow problems

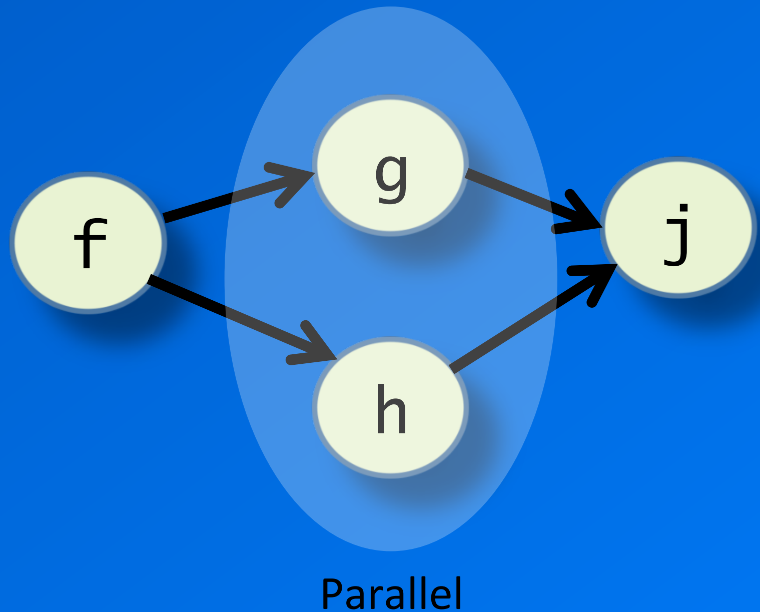
- Par really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input)
- Identify the nodes and edges of the graph
  - each node is created by fork
  - each edge is an IVar

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 ...
```



Implementation outline

- We need a type environment:

```
type TypeEnv = Map Var (IVar Type)
```

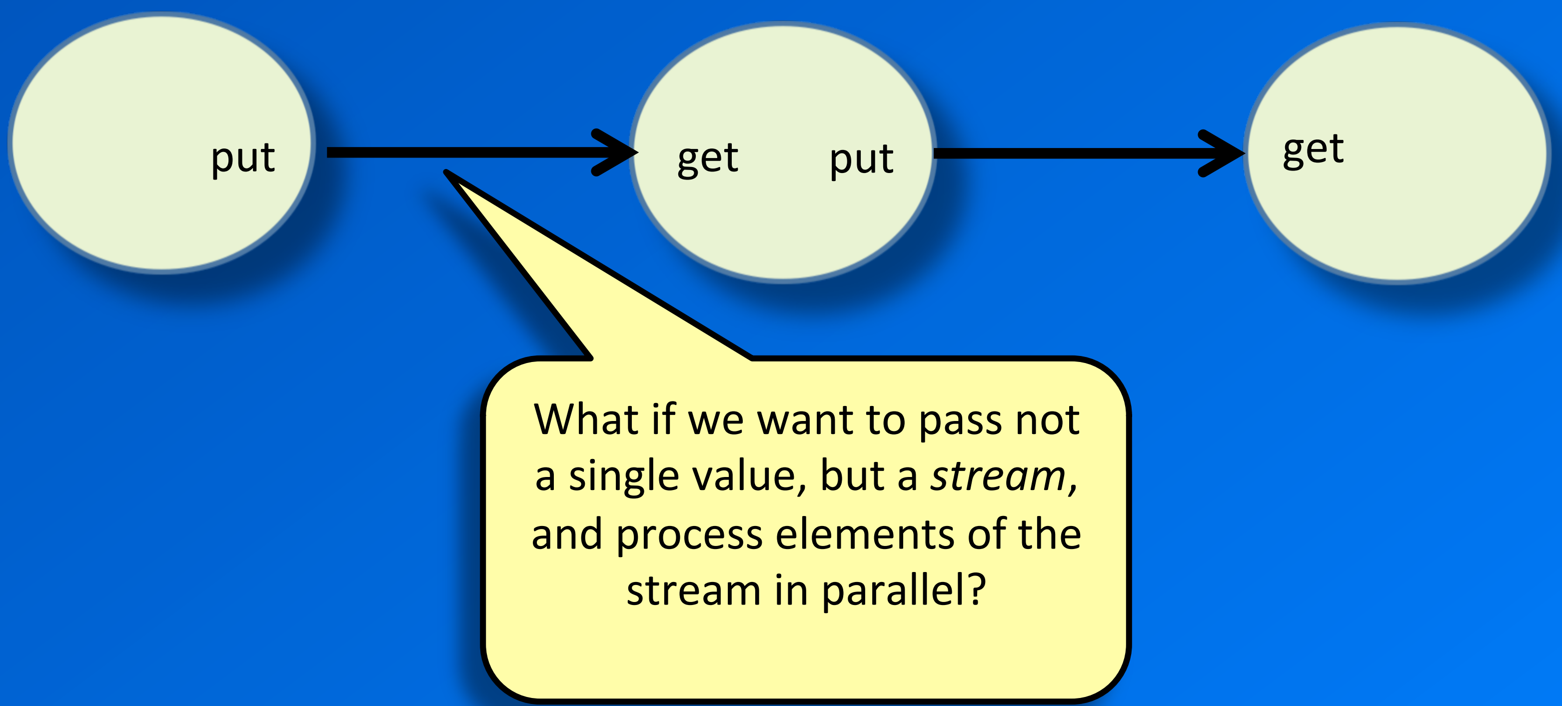
- To infer a type for a binding:
  - get the types of all its free variables
  - infer the type
  - put the result in the result IVar
- Do this for all the bindings in parallel, and Par will automatically take advantage of any parallelism
- (details in the book)

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

Pipeline parallelism



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
- 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 Stream.hs

Generate a Stream

- One way: generate a stream from a (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
```

Strict!

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

- 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
```

- Achieves 1.85 speedup vs. the sequential version on 2 cores (does not scale further)
- Another example (streaming RSA encoding/decoding) is in the sample code.

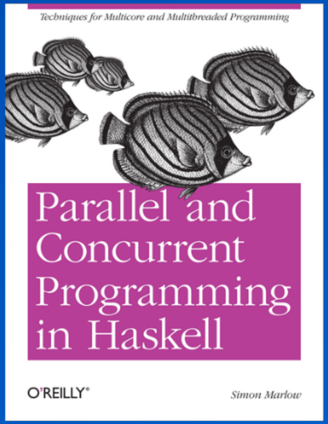
Limiting stream size

- What happens if the stream producer runs much faster than the stream consumer?
- Typically systems use some form of “backpressure” to solve this
- We can address this problem in the stream type:

```
data IList a = Null
             | Cons { hd :: a
                    , tl :: Stream a }
             | Fork (Par ())
                    (IList a)

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

Computation that creates more of the stream

Resources



`http://community.haskell.org/~simonmar/pcph/`

- These slides:

`http://community.haskell.org/~simonmar/Chalmers2014.pdf`

- Code samples

`https://github.com/simonmar/parconc-examples`

`http://hackage.haskell.org/package/parconc-examples`

```
cabal unpack parconc-examples
```

ThreadScope that works

```
$ git clone https://github.com/Mikolaj/ThreadScope.git
$ cd ThreadScope
$ cabal install
```