the Par Monad
Lecture 3

Mary Sheeran

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

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

http://hackage.haskell.org/packages/archive/monad-par-extras/0.3.2/doc/html/Control-Monad-Par-Combinator.html

Paper from Haskell'11

A Monad for Deterministic Parallelism

Simon Marlow
Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Ryan Newton
Intel, Hudson, MA, U.S.A
ryan.r.newton@intel.com

Simon Peyton Jones
Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com

Abstract

We present a new programming model for deterministic parallel computation in a pure functional language. The model is monadic and has explicit granularity, but allows dynamic construction of dataflow networks that are scheduled at runtime, while remaining deterministic and pure. The implementation is based on monadic concurrency, which has until now only been used to simulate concurrency in functional languages, rather than to provide parallelism. We present the API with its semantics, and argue that parallel execution is deterministic. Furthermore, we present a complete work-stealing scheduler implemented as a Haskell library, and we show that it performs at least as well as the existing parallel programming models in Haskell.

has explicit granularity, and uses I-structures [1] for communication. The monadic interface, with its explicit `fork` and communication, resembles a non-deterministic concurrency API; however by carefully restricting the operations available to the programmer we are able to retain determinism and hence present a pure interface, while allowing a parallel implementation. We give a formal operational semantics for the new interface.

Our programming model is closely related to a number of others; a detailed comparison can be found in Section 8. Probably the closest relative is pH [16], a variant of Haskell that also has I-structures; the principal difference with our model is that the monad allows us to retain referential transparency, which was lost in pH with the introduction of I-structures. The target domain of our programming model is large-grained irregular parallelism, rather than

builds on

[7] K. Claessen. A poor man's concurrency monad. Journal of Functional Programming, 9:313–323, May 1999.

In the beginning were

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

- pseq   expresses sequential evaluation order

+ par   turns a lazy computation into a future

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

Rules for par (from Par Monad paper)

You must

(a) pass an unevaluated computation to par

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

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

reasoning about par

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

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

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

Eval monad + Stratgies

The Eval monad

- simple primitives for introducing deterministic parallelism
- minimal control over the evaluation order (improvement over raw form of using par and pseq)

Strategies

- Adding parallelism over (lazy) data structures
- Composability: combine Strategies into larger ones
- Modularity: (e `using` s) separates the control of parallelism from the algorithm

But...

slide by Simon Marlow

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

Enter the Par Monad

From the Haskell'11 paper:

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

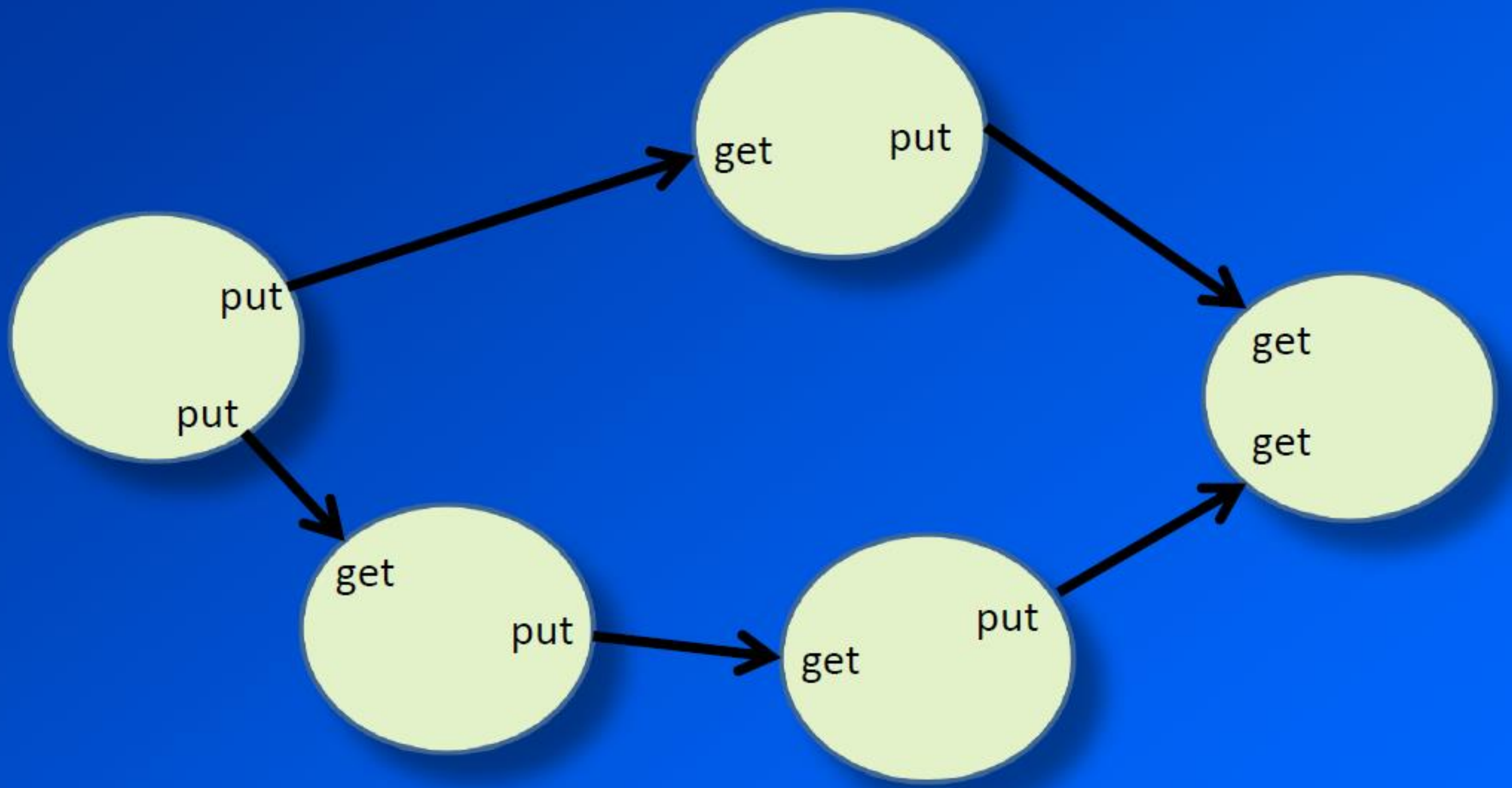
The Par Monad

- 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 () ->

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

Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

semantics of fork:

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

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a -> 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

The Par Monad

```
data Par
instance Monad Par

runPar :: Par a ->

fork :: Par ()

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

Par is a monad for parallel computation

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

results are communicated through IVars

IVar

write-once mutable reference cell

two operations, **`put`** and **`get`**

`put` assigns a value to the IVar.

`get` waits until the IVar has been assigned a value, and then returns the value

History: see I-structures (Arvind et al, 1989)
paper on course web page (notes for this lecture)

also pH (book by Nikhil and Arvind 2001, I don't have it ☹)

IV

interesting paper, 503 citations
builds on earlier work from 1981

write-once mutable refere

two operations, **`put`** and **`get`**

`put` assigns a value to the IVar.

`get` waits until the IVar has been assi… a value, and then returns the value

History: see I-structures (Arvind et al, 1989)
paper on course web page (notes for this lecture)

also pH (book by Nikhil and Arvind 2001, I don’t have it ☹)

put once

put ONCE per lvar

Later puts are runtime errors

This is necessary to preserve determinism

put strict

put is fully strict (fully evaluates its argument)

Can see this from the type

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

put strict

put is fully strict (fully evaluates its argument)

Can see this from the type

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

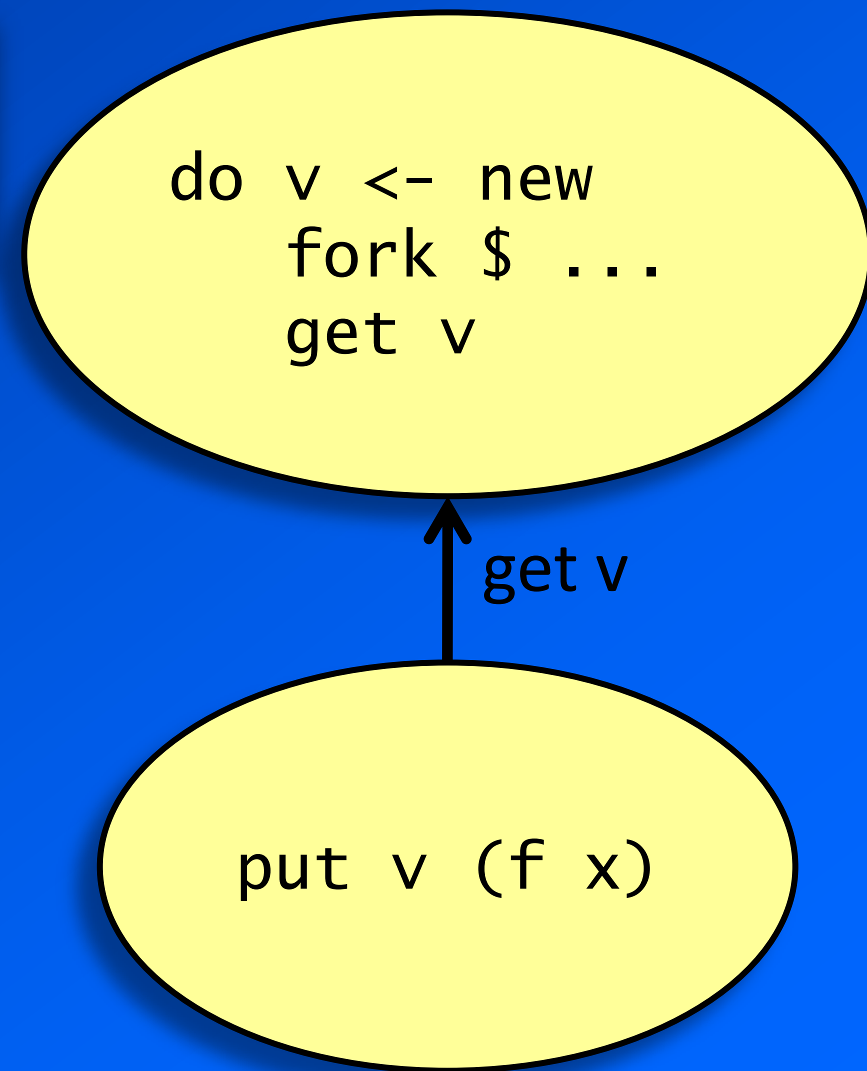
Idea : make it easy for the user to know when (in which thread) the work is done

things flowing along arcs in the data flow graph are fully evaluated

not allowed to put lazy computations into 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 (arrow) pointing from the node with the put in it to the node with the get in it

fork   creates a new node in the graph

get   creates a new edge (arrow) pointing from the node with the put in it to the node with the get in it

fork creates a new node in the graph

get c ointing from
t the
r

target domain:
large-grained irregular dynamic parallelism

put → get

fork   creates a new node in the graph

get   c … ointing from
t … the
r …

target domain:
large-grained irregular dynamic parallelism

fine-grained regular parallelism (data parallelism) comes later in the course
(see also DPH)

put → get

fork creates a new node in the graph

get c… ointing from

t… the

r…

target domain:
large-grained irregular dynamic parallelism

fine-grained regular parallelism (data parallelism) comes later in the course
(see also DPH)

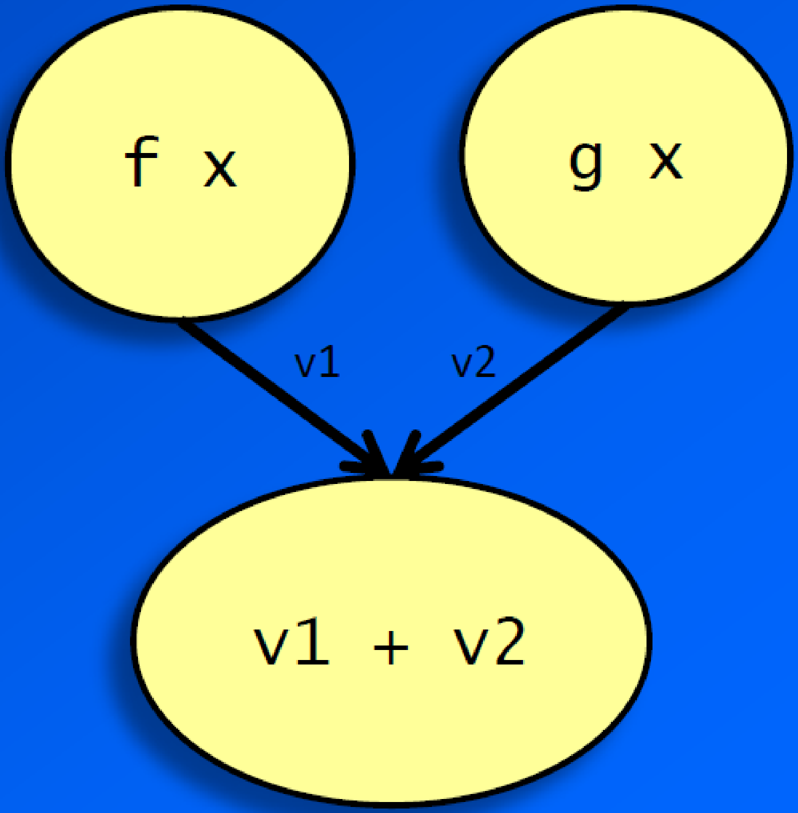
Note the Haskell approach of giving you a smörgåsbord of ways to do parallel programming

put … get

A bit more complex...

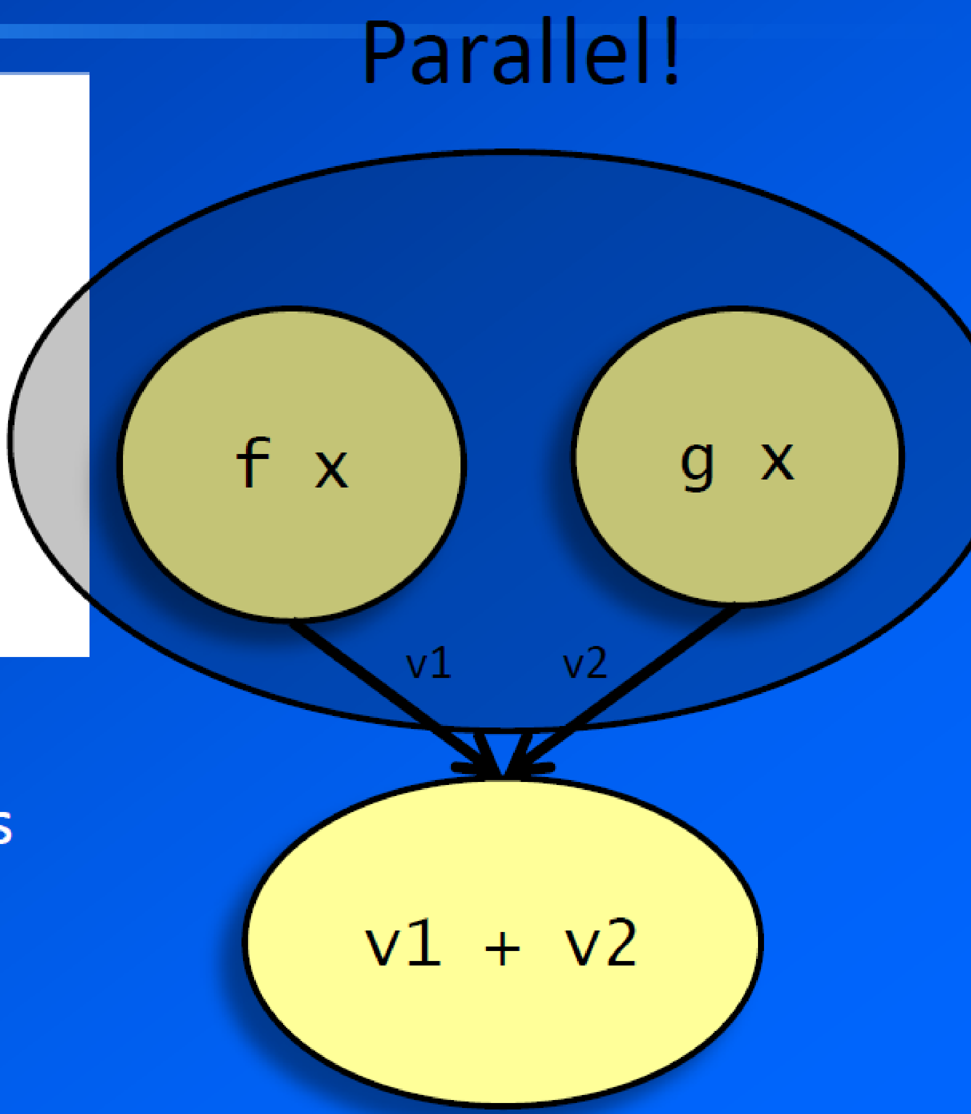
```
do v1 <- new
   v2 <- new
   fork $ put v1 (f x)
   fork $ put v2 (g x)
   get v1
   get v2
   return (v1 + v2)
```

- 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)
```

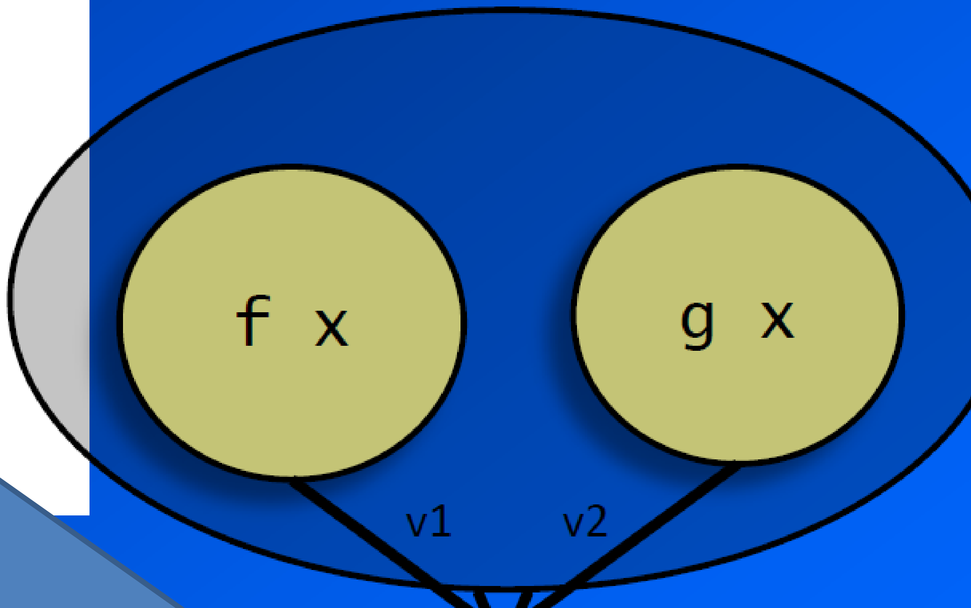


- runPar evaluates the graph
- nodes with no dependencies between them can execute in parallel

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)
```

Parallel!



- runPar evaluates the gra[…]
- nodes with no depe[…] between them can e[…] in parallel

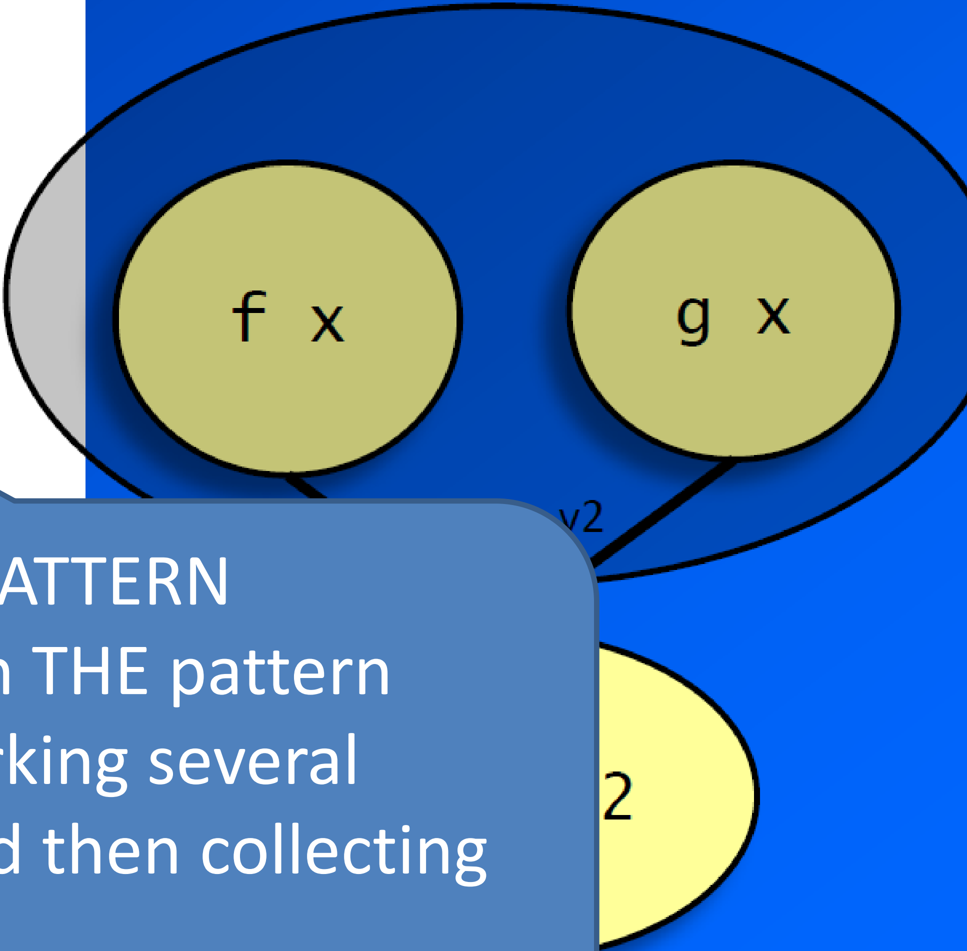
Note that put is fully strict
(=> normal form data NFData context)

Stuff flowing along arcs is fully evaluated

A bit more complex...

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

Parallel!



- runPar evaluat
- nodes with no between them in parallel

A PATTERN
maybe even THE pattern
a parent forking several children and then collecting results

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
```

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

or

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

to get Simon Marlow’s lecture notes plus code

The following example is

sudoku-par2.hs

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 ...
$
```

Slowdown on my laptop ☹

Using latest version of monad-par

Code>sudoku-par2 sudoku17.1000.txt +RTS -s -N2

….

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

```
INIT   time    0.00s  (  0.00s elapsed)
MUT    time    7.25s  (  3.60s elapsed)
GC     time    0.16s  (  0.10s elapsed)
EXIT   time    0.00s  (  0.00s elapsed)
Total  time    7.41s  (  3.70s elapsed)
```

Sequential version takes 1.82s

(Note that we are not using any sparks.)

No sp… ☹

Using latest

Code>sudoku-p

….

SPARKS: 0 (0 conve

INIT time 0.00s
MUT time 7.25
GC time 0.16s
EXIT time 0.00s
Total time 7.41s

Sequential version
(Note that we are no

couldn’t open the eventlog (out of memory)

Made smaller 200 prob ex.
Had 2.7 million events!

Consulted Simon Marlow who diagnosed a problem with the current ”direct” scheduler

Got workaround ☺

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

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')
```

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

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

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

    print $ length $ filter i
       i1 <- new
       i2 <- new
       fork $ put i1 (map so
       fork $ put i2 (map so
       as' <- get i1
       bs' <- get i2
       return (as' ++ bs')
```

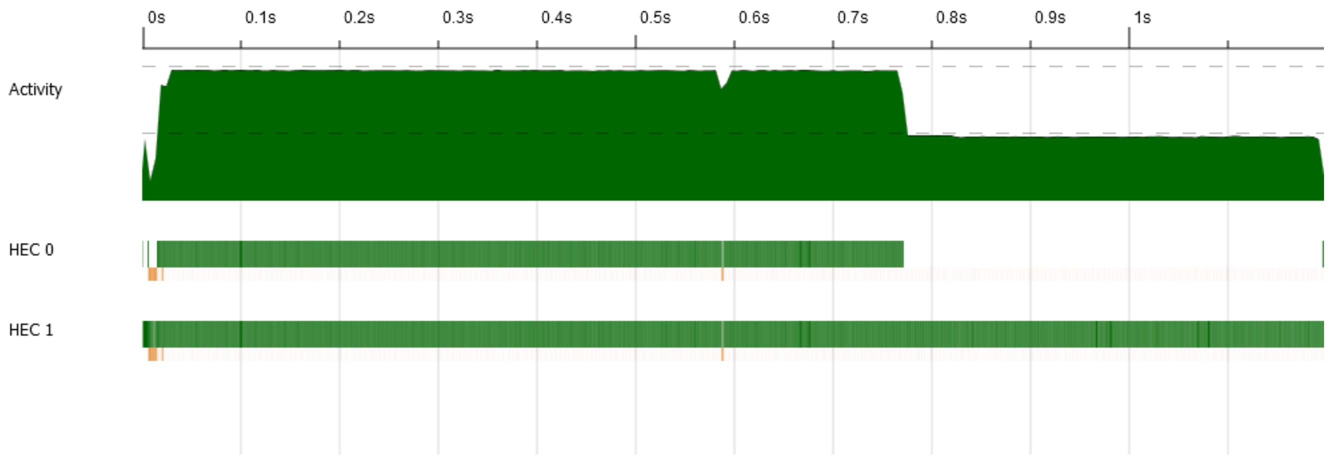
Reverts to a different default scheduler

Speedup after all

```
Sequential   sudoku-par1

Code>sudoku-par1 sudoku17.1000.txt +RTS  -s
1000
  . . .
 Total   time    1.81s  (  1.82s elapsed)
```

```
Code>sudoku-par2 sudoku17.1000.txt +RTS -N2 -s
1000
  . . .

 SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
  . . .
 Total   time    1.94s  (  1.19s elapsed)
```

Are we happy?

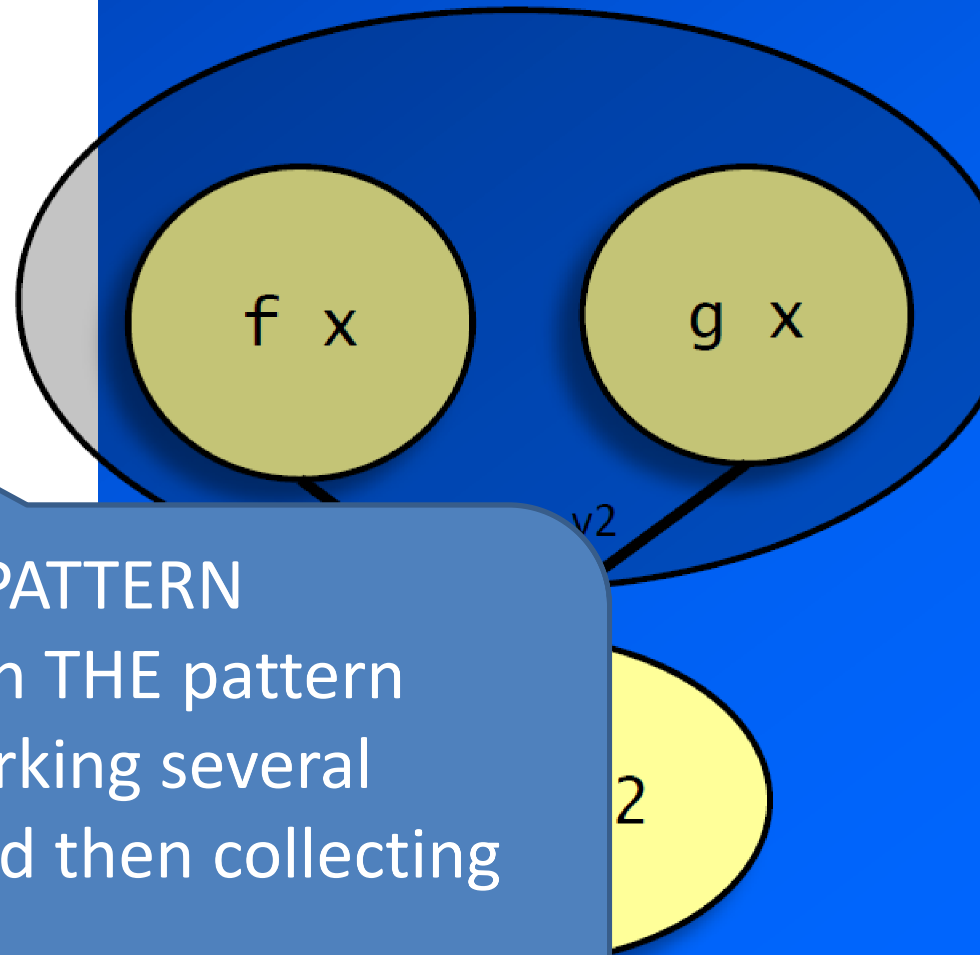


A bit more complex...

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

- runPar evaluat
- nodes with no
  between them
  in parallel

Parallel!



A PATTERN
maybe even THE pattern
a parent forking several
children and then collecting
results

Capture it

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

Capture it

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

or

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

Capture it

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

or

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

(>>=) :: Monad m => m a -> (a -> m b) -> m b

what are types of p put r ?

Capture it

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

or

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

(>>=) :: Monad m => m a -> (a -> m b) -> m b

what are types of p put r ?

Par a a -> Par ()

Capture it

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

First one child

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

Capture it

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

spawn subsumes fork,new,put

prevents errors involving too many puts (runtime errors)

still sometimes want to use fork etc

Capture it

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

and to spawn a pure (rather than monadic) computation

```
spawnP :: NFData a => a -> Par (IVar a)
spawnP = spawn . return
```

parMap

```
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
parMap f xs = mapM (spawnP . f) xs >>= mapM get
```

or

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

parMap

```
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
parMap f xs = mapM (spawnP . f) xs >>= mapM get
```

or

```
parMap :: NFData b =>
parMap f xs = do
  ibs <- mapM (spawnP
  mapM get ibs
```

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

parMap

```
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
parMap f xs = mapM (spawnP . f) xs >>= mapM get
```

or

```
parMap :: NFData b =>
parMap f xs = do
  ibs <- mapM (spawnP
  mapM get ibs
```

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

Here, f is pure

version with monadic f

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f xs = mapM (spawn . f) xs >>= mapM get
```

or

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

version with monadic f

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f xs = mapM (spawn . f) xs >>= mapM get
```

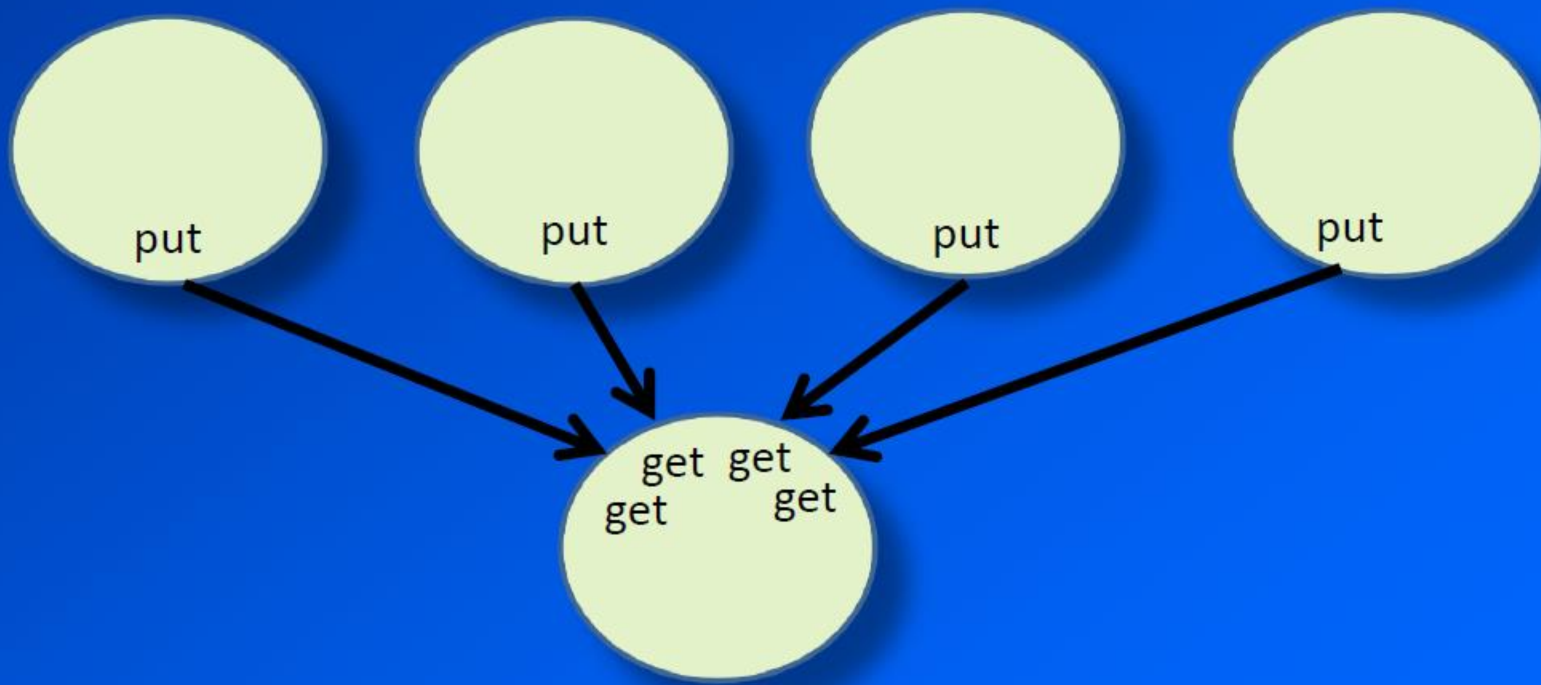
or

```
parMapM :: N
parMapM f xs
  ibs <- map
  mapM get i
```

Versions of parMap and parMapM in library work for any Traversble data structure, not just lists

Defined in Control.Monad.Par.Combinator

What is the dataflow graph?



Sudoku-par3.hs

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

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

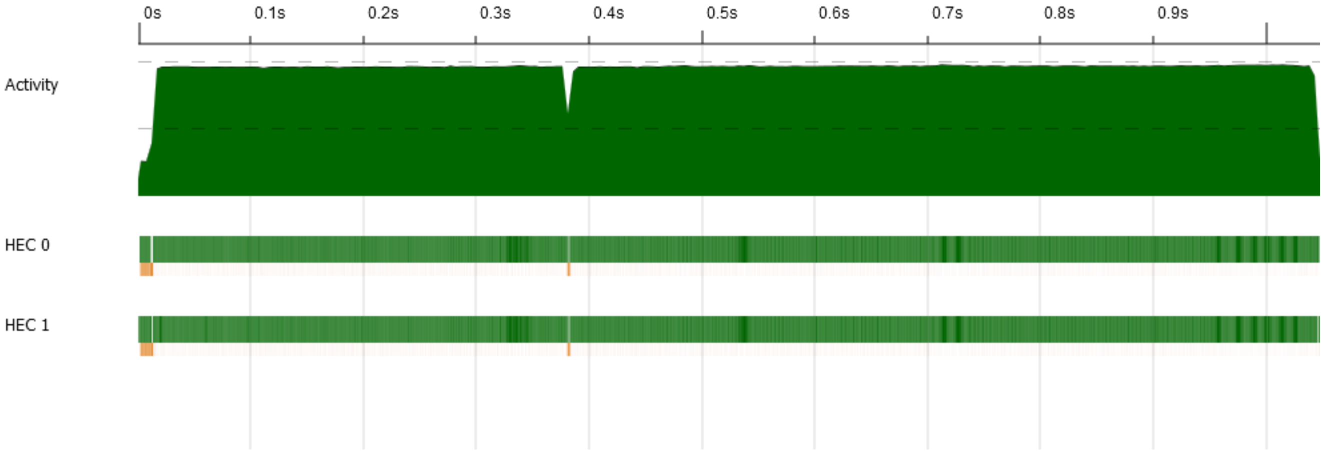
Performance?

```
C:\Users\Ms\Programs\PFP Code>sudoku-par3 sudoku17.1000.txt +RTS -N2 -s
1000

  . . .

  Total   time    1.94s  (  0.97s elapsed)
```

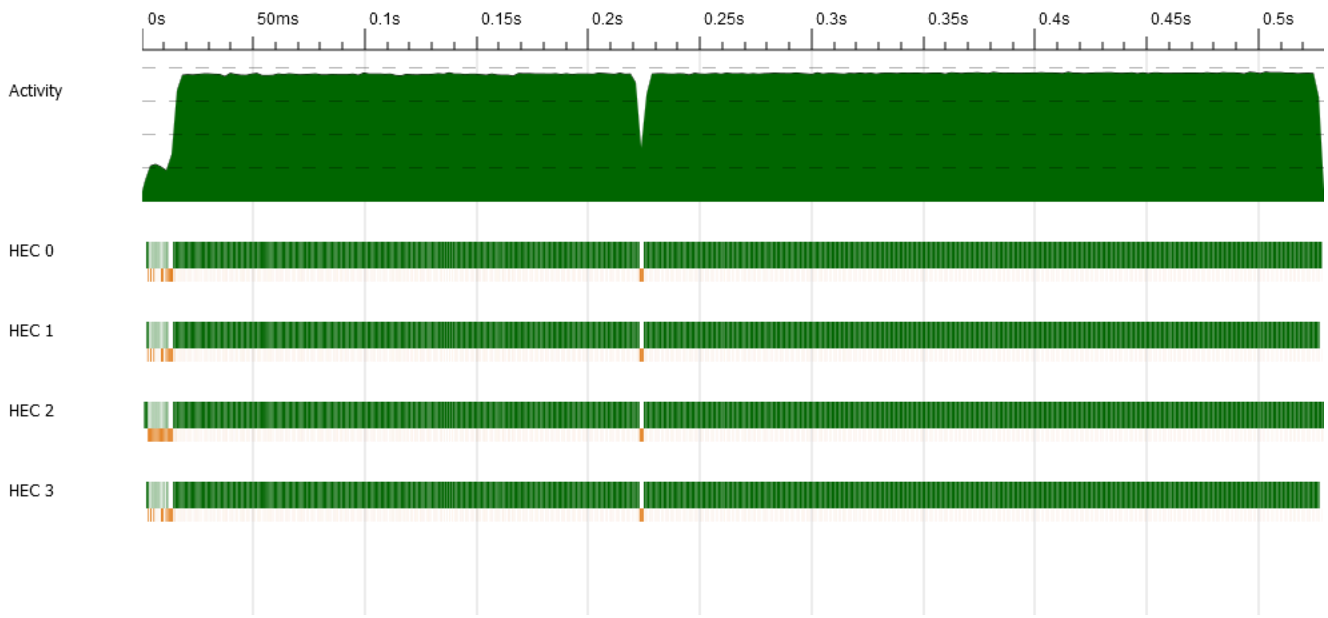
Speedup calculated from the sequential timing (not from –N1)

1.82 / 0.97 = 1.87 ☺

Looks better too



and scales



0s
50ms
0.1s
0.15s
0.2s
0.25s
0.3s
0.35s
0.4s
0.45s
0.5s
Activity
HEC 0
HEC 1
HEC 2
HEC 3

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)

Sudoku-par4.hs

```
import Control.Exception
import System.Environment
import Data.Maybe
import Control.DeepSeq
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par.Combinator

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

Results
(16000 puzzles)

no chunks (sudoku-par3) sequential
Total time 27.09s (27.13s elapsed)

no chunks (sudoku-par3) –N4
Total time 33.66s (8.46s elapsed)

chunk 10 –N4
Code>sudoku-par4 sudoku17.16000.txt 10 +RTS -N4 -s
Total time 32.72s (8.21s elapsed)

chunk 100 –N4
Total time 30.48s (7.69s elapsed)

chunk 200 –N4
Total time 29.62s (7.60s elapsed) best speedup 3.57

chunk 1000 –N4
Total time 32.61s (8.58s elapsed)

Another pattern D&C

```
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)
```

merge sort

```
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) = splitAt (div (length xs + 1) 2) xs
```

merge sort

```
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, b
      where (as,bs) = splitAt (div (l          + 1) 2) xs
```

sequential merge

sort is sequential sort from Data.List

merge sort

```
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) = splitAt (div (length xs + 1) 2) xs
```

”prob” is (Int,[Integer])

(threshold,list)

Results

on 200k list of Integers (from last year's sorting competition)

sequential: sort from Data.List 401ms

parallel (threshold 12 in all cases)

-N1 396ms
-N2 279ms
-N4 215ms

not bad!

Dataflow problems

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

slide by Simon Marlow

Larger examples to study

parallel type inferencer (see Haskell'11 paper)

k-means (see Marlow's lecture notes)

Related work (Par Monad, see paper)

- fork / join   Habanero Java, Cilk
- sync. data structures   pH, concurrent ML
- Manticore supports both CML model and explict futures
- Intel Concurrent Collections (CnC) provide a superset of Par Monad functionality

Challenge

Look in
`Control.Monad.Par.Combinator`

It contains a few combinators, but needs more!

Design and implement some new combinators.

Challenge

Look in
`Control.Monad.P`

It contains a few comb

Design and implement so ew combinators.

If we like your proposal enough, we'll send it to Simon Marlow and Ryan Newton to see if they like it too (and want to include it)

(will be optional part of a lab later)

Final words on Par

- runPar is more costly than runEval (but still fairly cheap)
- puts its faith in higher-order skeletons as the means to provide modular parallelism
- Friday’s lecture: Kevin Hammond (co-author on first Strategies paper) on high-level structured parallel programming
- lecture on skeletons by Jost Berthold the following week

Final words on Par

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

Final words on Par

- Par Monad scheduler separate from runtime, easily changed
- Perhaps ordinary mortals should use Par, while par is used for automated parallelisation??
- See Lennart Augustsson's Report from the Real World on May 2. He will likely return to the strict vs lazy question (or rather to the question of controlling evaluation)

Open research problems?

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

slide by Simon Marlow