Parallel Functional Programming Lecture 3

Mary Sheeran

(with thanks to Simon Marlow for use of slides)

https://chalmers.instructure.com/courses/23443

Remember nfib

```
nfib :: Integer -> Integer
nfib n | n<2 = 1
nfib n = nfib (n-1) + nfib (n-2) + 1
```

- A trivial function that returns the number of calls made—and makes a very large number!

n	nfib n
10	177
20	21891
25	242785
30	2692537

Parallelism

par x y

- ”Spark” x in parallel with computing y
  - (and return y)
- The run-time system *may* convert a spark into a parallel task—or it may not
- Starting a task is cheap, but not free

Parallelism

x `par` y

Sequencing

`pseq x y`

- Evaluate x *before* y (and return y)
- Used to *ensure* we get the right evaluation order

Sequencing

```
x `pseq` y
```

- Binds more tightly than par

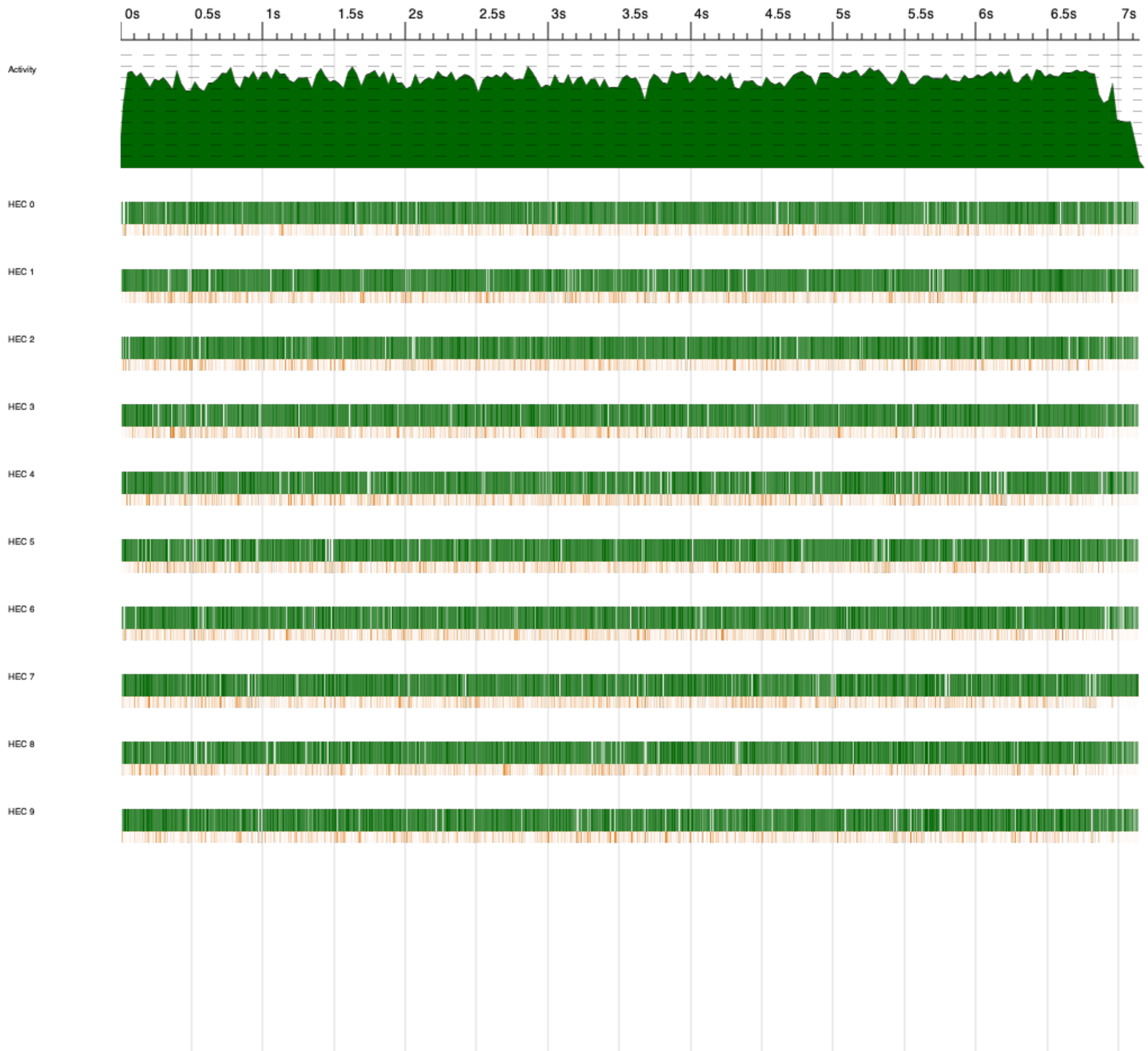
Using par and pseq

```
import Control.Parallel

rfib :: Integer -> Integer
rfib n | n < 2 = 1
rfib n = nf1 `par` (nf2 `pseq` nf2 + nf1 + 1)
  where nf1 = rfib (n-1)
        nf2 = rfib (n-2)
```

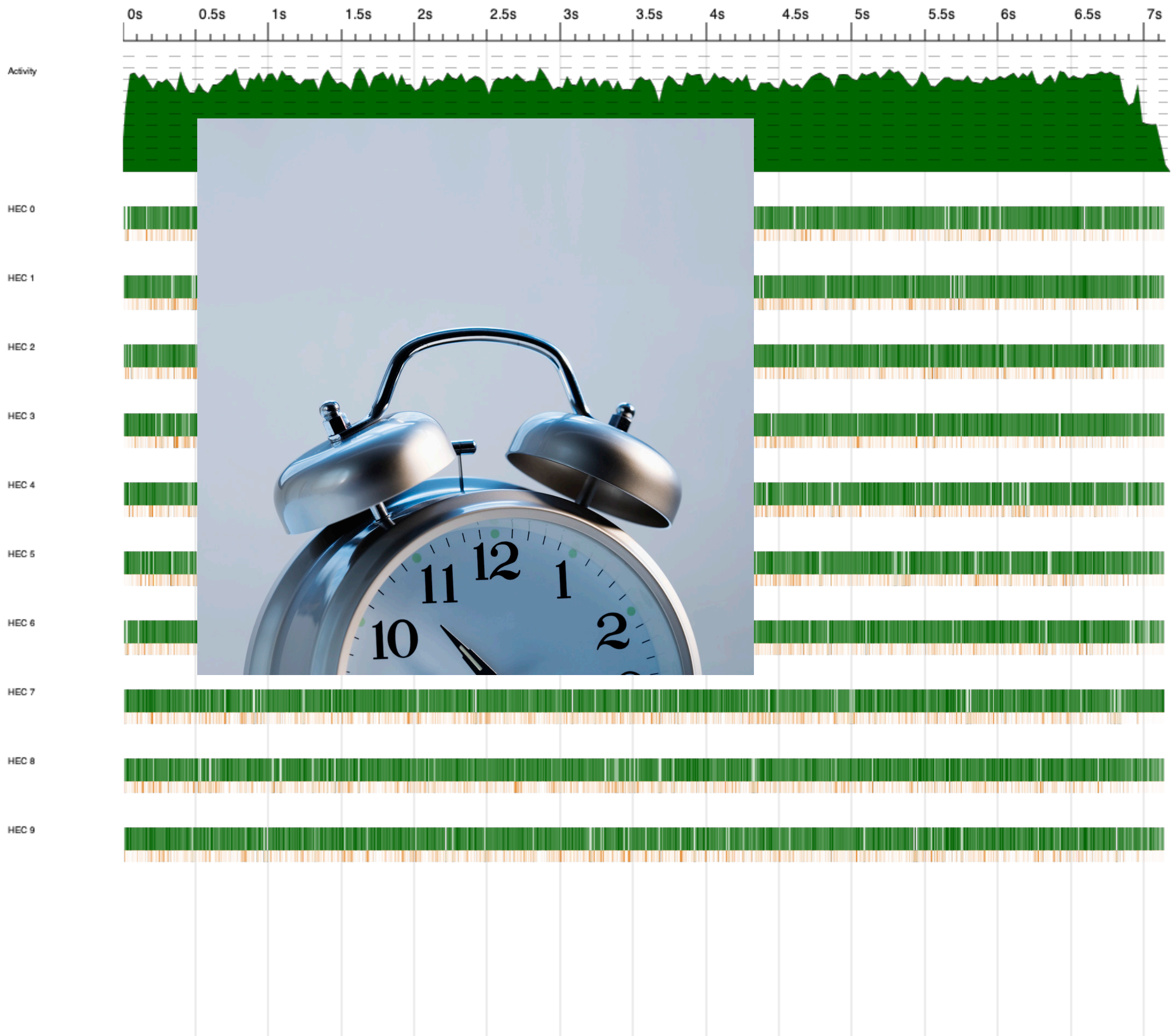
Using par and pseq

```
import Control.Parallel

rfib :: Integer -> Integer
rfib n | n < 2 = 1
rfib n = nf1 `par` (nf2 `pseq` nf2 + nf1 + 1)
  where nf1 = rfib (n-1)
        nf2 = rfib (n-2)
```

- Evaluate nf1 *in parallel with (*Evaluate nf2 *before* …)



0s
0.5s
1s
1.5s
2s
2.5s
3s
3.5s
4s
4.5s
5s
5.5s
6s
6.5s
7s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
HEC 6
HEC 7
HEC 8
HEC 9


0s
0.5s
1s
1.5s
2s
2.5s
3s
3.5s
4s
4.5s
5s
5.5s
6s
6.5s
7s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
HEC 6
HEC 7
HEC 8
HEC 9

What's happening?

```
$stack run -- +RTS -N10 -s
```

-s   to get stats

Hah

```
331160281

…

SPARKS: 165597162 (7874 converted, 94153001 overflowed,
0 dud, 7257490 GC'd, 37481069 fizzled)


  INIT    time    0.000s  (  0.007s elapsed)
  MUT     time   17.660s  (  2.256s elapsed)
  GC      time    0.099s  (  0.097s elapsed)
  EXIT    time    0.000s  (  0.010s elapsed)
  Total   time   17.760s  (  2.371s elapsed)
```

Hah

```
331160281

…

SPARKS: 165597162 (7874 converted, 94153001 overflowed,
0 dud, 7257490 GC'd, 37481069 fizzled)

  INIT    time    0.000s
  MUT     time   17.660s
  GC      time    0.099s
  EXIT    time    0.000s
  Total   time   17.760s  (  2.371s elapsed)
```

converted = turned into useful parallelism

Controlling Granularity

- Let's use a threshold for going sequential, t

```
tfib :: Integer -> Integer -> Integer
tfib t n | n < t = sfib n
tfib t n = nf1 `par` nf2 `pseq` nf1 + nf2 + 1
  where nf1 = tfib t (n-1)
        nf2 = tfib t (n-2)
```

Better

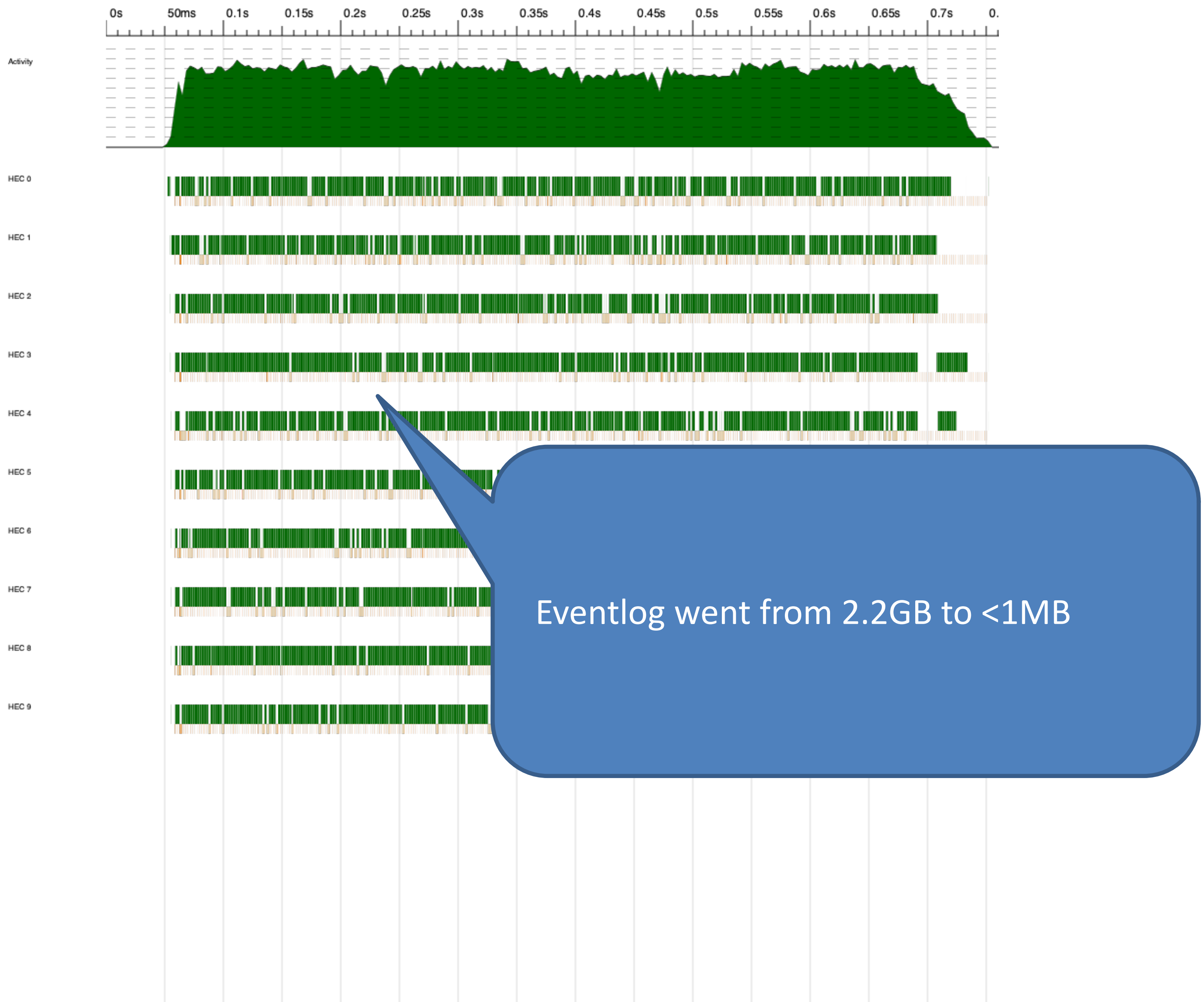
```
tfib 32 40      gives

SPARKS: 88 (63 converted, 0 overflowed, 0 dud,
 0 GC'd, 25 fizzled)

  INIT    time    0.000s  (  0.006s elapsed)
  MUT     time    5.044s  (  0.648s elapsed)
  GC      time    0.024s  (  0.029s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time    5.068s  (  0.685s elapsed)
```

Better

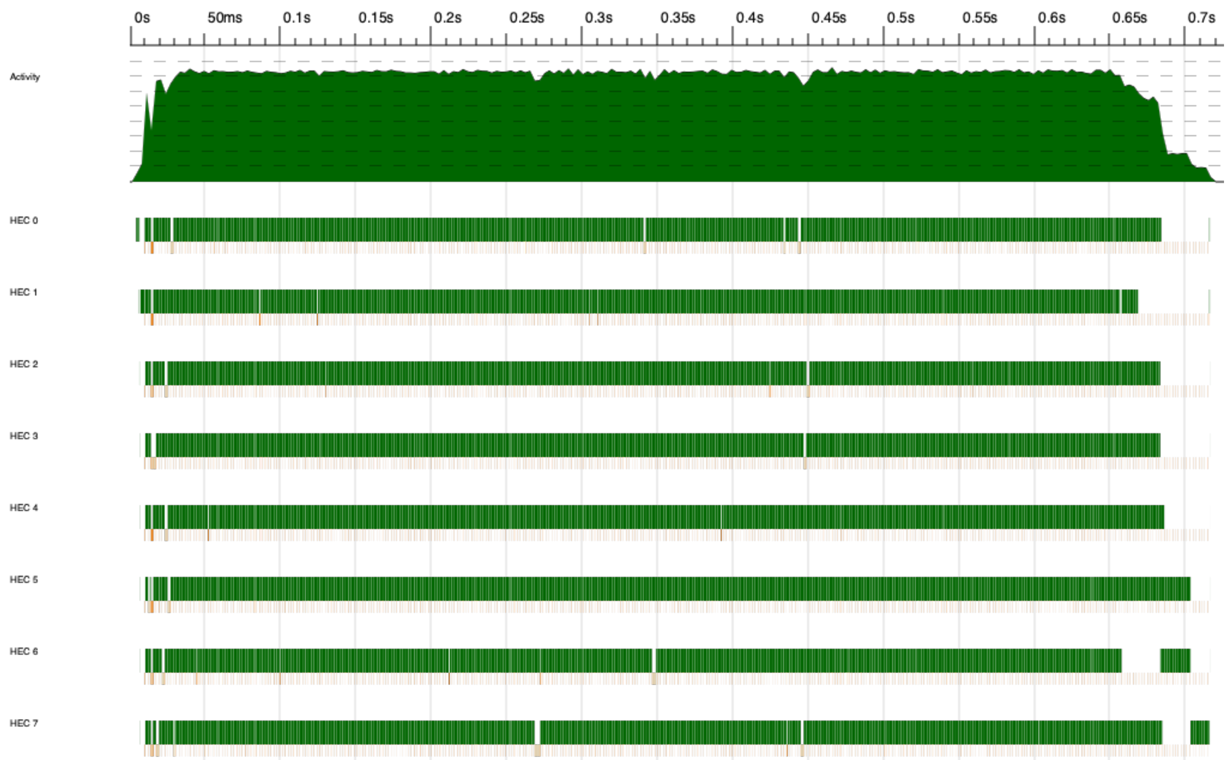
```
tfib 32 40      gives

SPARKS: 88 (63 converted, 0 overflowed, 0 dud,
 0 GC'd, 25 fizzled)
```

Was 1.7s on my previous (4 core) Mac ☺

```
  INIT    time    0                              d)
  MUT     time    5                              d)
  GC      time    0.                            ed)
  EXIT    time    0.000s  (  0.0   s elapsed)
  Total   time    5.068s  (  0.685s elapsed)
```


0s
50ms
0.1s
0.15s
0.2s
0.25s
0.3s
0.35s
0.4s
0.45s
0.5s
0.55s
0.6s
0.65s
0.7s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
HEC 6
HEC 7
HEC 8
HEC 9
Eventlog went from 2.2GB to <1MB

8 cores slightly better (efficiency cores perhaps not so useful)



30413 events, 0.726s

(next step is to move to Criterion, then no eventlog, get slightly better speed)

What are we controlling?

The division of the work into possible parallel tasks (par) including choosing size of tasks

GHC runtime takes care of choosing which sparks to actually evaluate in parallel and of distribution (load balancing)

Need also to control order of evaluation (pseq) and degree of evaluation

Dynamic behaviour is the term used for how a pure function gets partitioned, distributed and run

Remember, this is deterministic parallelism. The answer is always the same!

positive so far (par and pseq)

Don’t need to

- express communication
- express synchronisation
- deal with threads explicitly

BUT

par and pseq are difficult to use ☹

BUT

par and pseq are difficult to use ☹

MUST

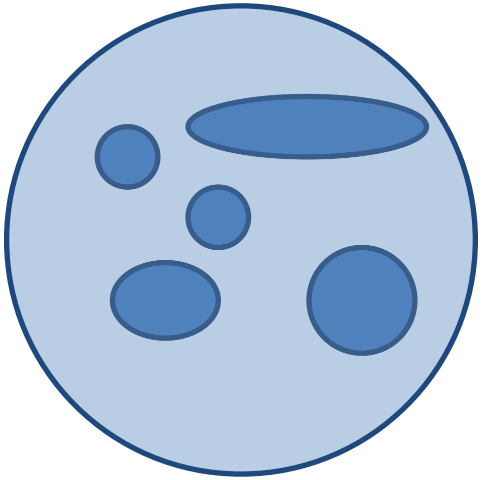
Pass an unevaluated computation to par

It must be somewhat expensive

Make sure the result is not needed for a bit

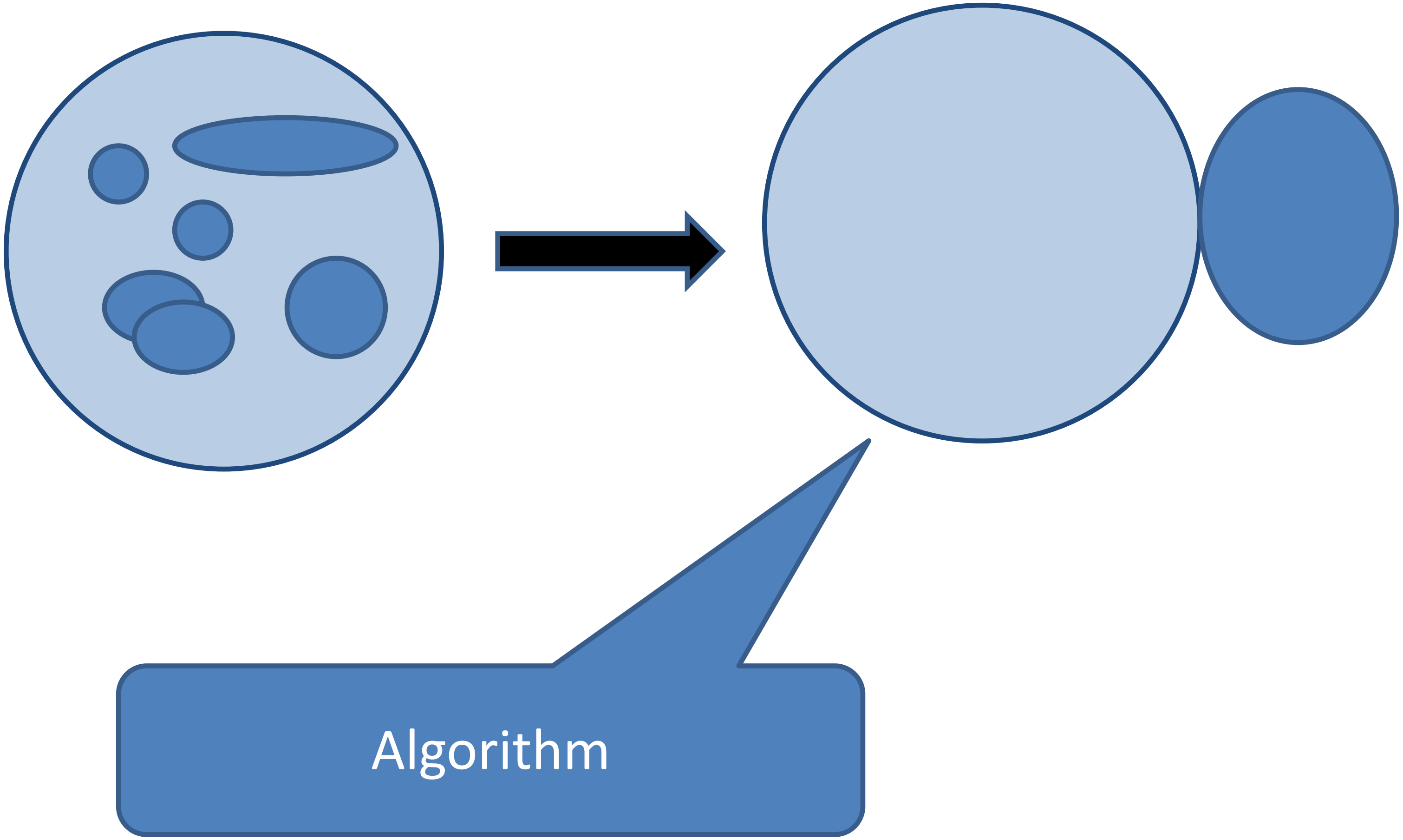
Make sure the result is shared by the rest of the program

Even if you get it right

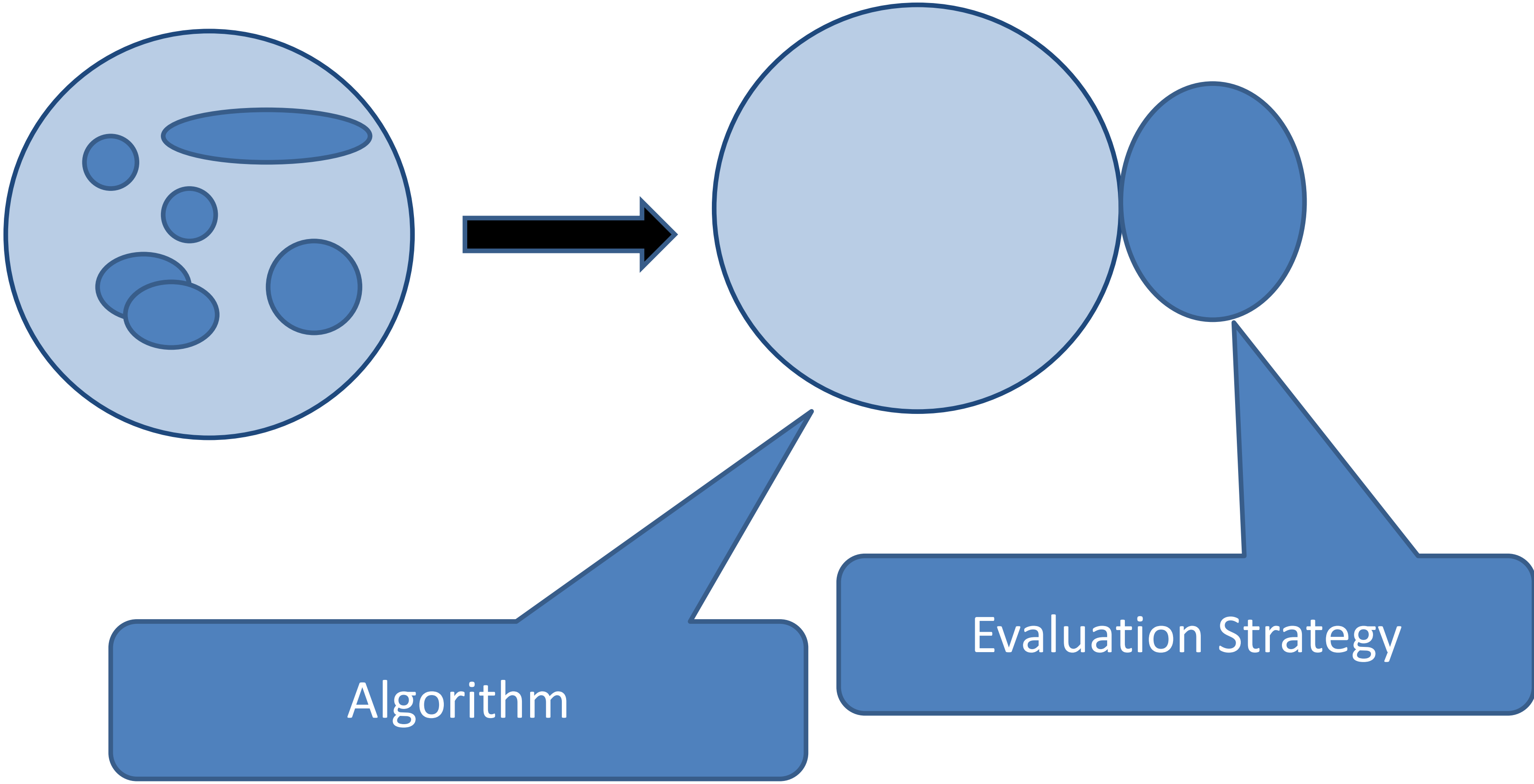


Original code + par + pseq + rnf etc.
can be opaque

Separate concerns



Separate concerns



Evaluation Strategies

express dynamic behaviour independent of the algorithm

provide abstractions above par and pseq

are modular and compositional
(they are ordinary higher order functions)

can capture patterns of parallelism

Papers (last lecture)

Haskell on a Shared-Memory Multiprocessor

Tim Harris Simon Marlow Simon Peyton Jones
Microsoft Research, Cambridge
{tharris,simonmar,simonpj}@microsoft.com

Haskell'05

Runtime Support for Multicore Haskell

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

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

Satnam Singh
Microsoft Research
Cambridge, U.K.
satnams@microsoft.com

ICFP'09

Papers (last lecture)

Haskell on a Shared-Memory Multiprocessor

Tim Harris Simon Marlow Simon Peyton Jones
Microsoft Research, Cambridge
{tharris,simonmar,simonpj}@microsoft.com

Haskell'05

85

Runtime Support for Multicore Haskell

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

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

Satnam Singh
Microsoft Research
Cambridge, U.K.
satnams@microsoft.com

ICFP'09

Papers (last lecture)

Haskell on a Shared-Memory Multiprocessor

Tim Harris Simon Marlow Simon Peyton Jones
Microsoft Research, Cambridge
{tharris,simonmar,simonpj}@microsoft.com

Haskell'05

85

Runtime Support for Multicore Haskell

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

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

Satnam Singh
Microsoft Research
Cambridge, U.K.
satnams@microsoft.com

ICFP'09

183

Latter won ICFP 10 year most influential paper
=> a retrospective blog post

https://blog.sigplan.org/2019/12/16/runtime-support-for-multicore-haskell-a-retrospective/

Papers

Algorithm + Strategy = Parallelism

P.W. TRINDER
Department of Computing Science, University of Glasgow, Glasgow, UK

K. HAMMOND
Division of Computing Science, University of St Andrews, St Andrews, UK

H.-W. LOIDL AND S.L. PEYTON JONES †
Department of Computing Science, University of Glasgow, Glasgow, UK

JFP 1998

Seq no more: Better Strategies for Parallel Haskell

Simon Marlow
Microsoft Research, Cambridge, UK
simonmar@microsoft.com

Patrick Maier
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University, Edinburgh, UK
H.W.Loidl@hw.ac.uk

Mustafa K. Aswad
Heriot-Watt University, Edinburgh, UK
mka19@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.uk

Haskell'10

Papers

Algorithm + Strategy = Parallelism

P.W. TRINDER
Department of Computing Science, University of Glasgow, Glasgow, UK

K. HAMMOND
Division of Computing Science, University of St Andrews, St Andrews, UK

H.-W. LOIDL AND S.L. PEYTON JONES †
Department of Computing Science, University of Glasgow, Glasgow, UK

JFP 1998

389

Seq no more: Better Strategies for Parallel Haskell

Simon Marlow
Microsoft Research, Cambridge, UK
simonmar@microsoft.com

Patrick Maier
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University, Edinburgh, UK
H.W.Loidl@hw.ac.uk

Mustafa K. Aswad
Heriot-Watt University, Edinburgh, UK
mka19@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.uk

Haskell'10

Papers

Algorithm + Strategy = Parallelism

P.W. TRINDER
Department of Computing Science, University of Glasgow, Glasgow, UK
K. HAMMOND
Division of Computing Science, University of St Andrews, St Andrews, UK
H.-W. LOIDL AND S.L. PEYTON JONES †
Department of Computing Science, University of Glasgow, Glasgow, UK

JFP 1998

389

Seq no more: Better Strategies for Parallel Haskell

Simon Marlow
Microsoft Research, Cambridge, UK
simonmar@microsoft.com

Patrick Maier
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University, Edinburgh, UK
H.W.Loidl@hw.ac.uk

Mustafa K. Aswad
Heriot-Watt University, Edinburgh, UK
mka19@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.uk

109

Haskell'10

Papers

Algorithm + Strategy = Pa

P.W. TRINDER
Department of Computing Science, University of Glas

K. HAMMOND
Division of Computing Science, University of St Andrew

H.-W. LOIDL AND S.L. PEYTON J
Department of Computing Science, University of Glas

Redesigns strategies

richer set of parallelism combinators
Better specs (evaluation order)
Allows new forms of coordination
generic regular strategies over data structures
speculative parellelism
based on a simple monad

Presentation is about New Strategies

Seq no more: Bette

Haskell 10

Simon Marlow
Microsoft Research, Cambridge, UK
simonmar@microsoft.com

Patrick Maier
-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Heriot-Watt University, Edinburgh, UK
H.W.Loidl@hw.ac.uk

Mustafa K
Heriot-Watt Univ
mka19@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.uk

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)

Slide borrowed from Simon Marlow’s CEFP slides, with thanks

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
         nf1 <- rpar (qfib (n-1))
         nf2 <- rseq (qfib (n-2))
         return (nf1 + nf2 + 1)
```

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
           nf1 <- rpar (qfib (
           nf2 <- rseq (qfib (
           return (nf1 + nf2 +
```

do this
spark qfib (n-1)

"My argument could be evaluated in parallel"

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
         nf1 <- rpar (qfib (
         nf2 <- rseq (qfib (
         return (nf1 + nf2 +
```

do this
spark qfib (n-1)

"My argument could be evaluated in parallel"

Remember that the argument should be a thunk!

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
           nf1 <- rpar (qfib (n-1))
           nf2 <- rseq (qfib (n-2))
           return (nf1 + nf2 + 1)
```

and then this
Evaluate qfib(n-2)
and wait for
result

"Evaluate my argument and wait for the result."

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
            nf1 <- rpar (qfib (n-1))
            nf2 <- rseq (qfib (n-2))
            return (nf1 + nf2 + 1)
```

the result

Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
           nf1 <- rpar (qfib (n-1))
           nf2 <- rseq (qfib (n-2))
           return (nf1 + nf2 + 1)
```

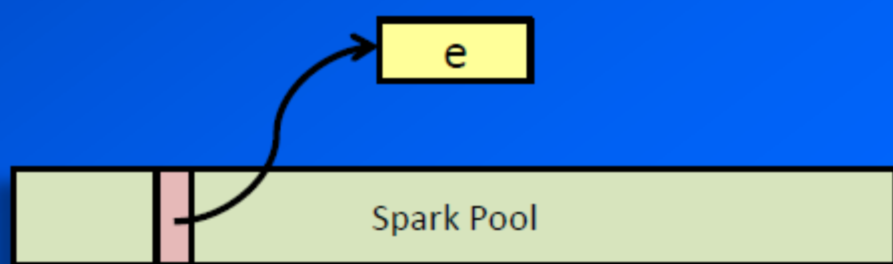
pull the answer out of the monad

(should add granularity control too!)

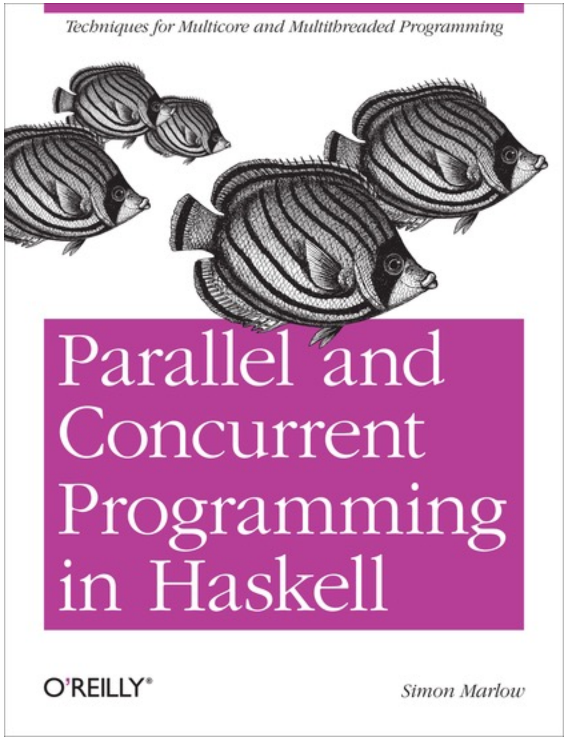
What does rpar *actually do?*

```
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!



Slide borrowed from Simon Marlow's CEFP slides, with thanks



Read Chapters 2 and 3

What do we have?

The Eval monad raises the level of abstraction for pseq and par; it makes fragments of evaluation order first class, and lets us compose them together. We should think of the Eval monad as an Embedded Domain-Specific Language (EDSL) for expressing evaluation order, embedding a little evaluation-order constrained language inside Haskell, which does not have a strongly-defined evaluation order.

(from Haskell 10 paper)

a possible parallel map

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

a possible parallel map

```
import Control.Parallel.Strategies

foo :: Integer -> Integer
foo a = sum [1 .. a]

main
  = print $ sum $ runEval $
         pMap foo (reverse [1..10000])
```

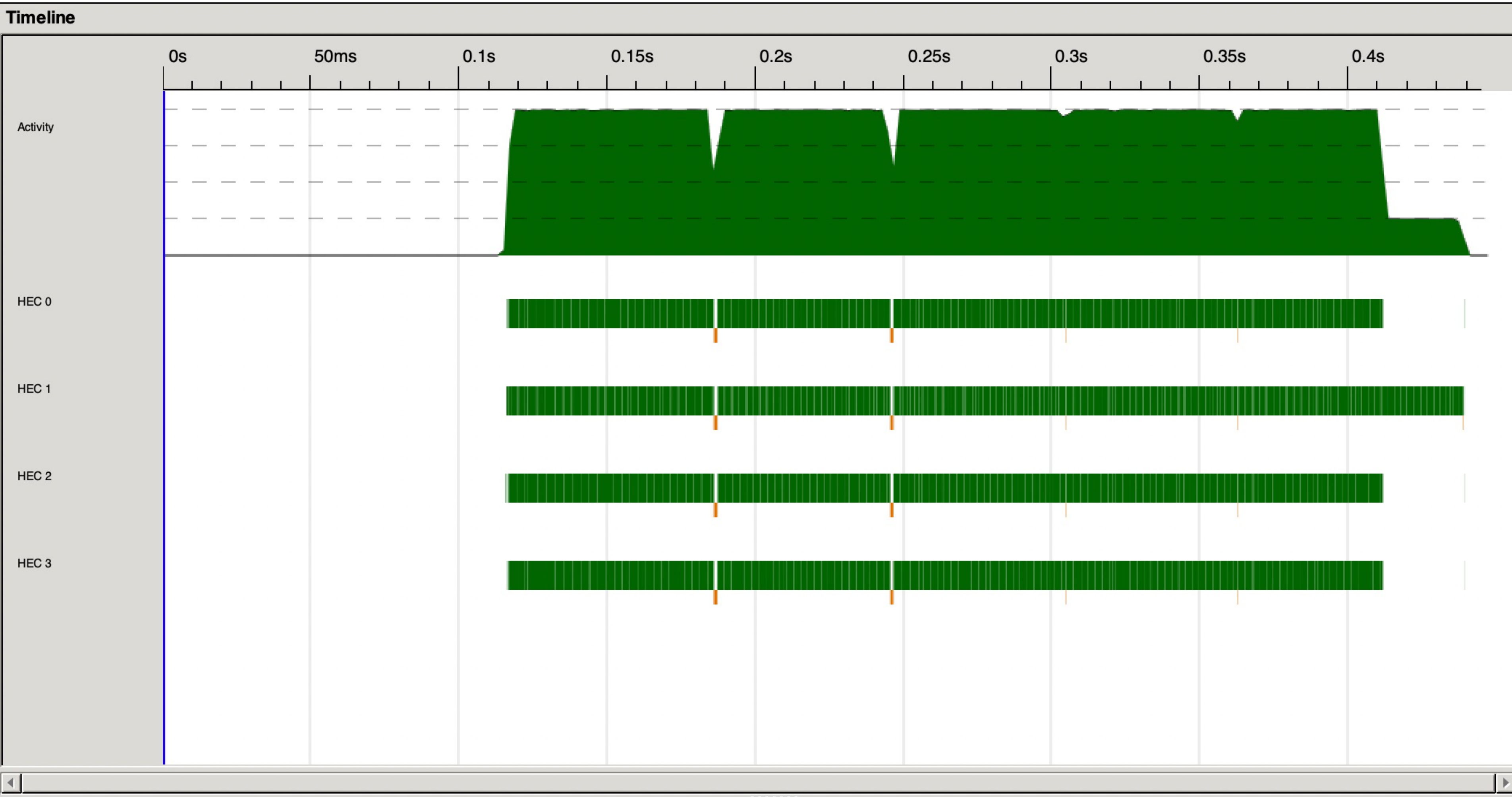
SPARKS: 10000 (8195 converted, 1805 overflowed, 0 dud, 0 GC'd, 0 fizzled)

```
INIT    time    0.003s  (  0.009s elapsed)
MUT     time    1.346s  (  0.410s elapsed)
GC      time    0.010s  (  0.003s elapsed)
EXIT    time    0.001s  (  0.000s elapsed)
Total   time    1.361s  (  0.423s elapsed)
```

#sparks = length of list

SPARKS: 10000 (8195 converted, 1805 overflowed, 0 dud, 0 GC'd, 0 fizzled)

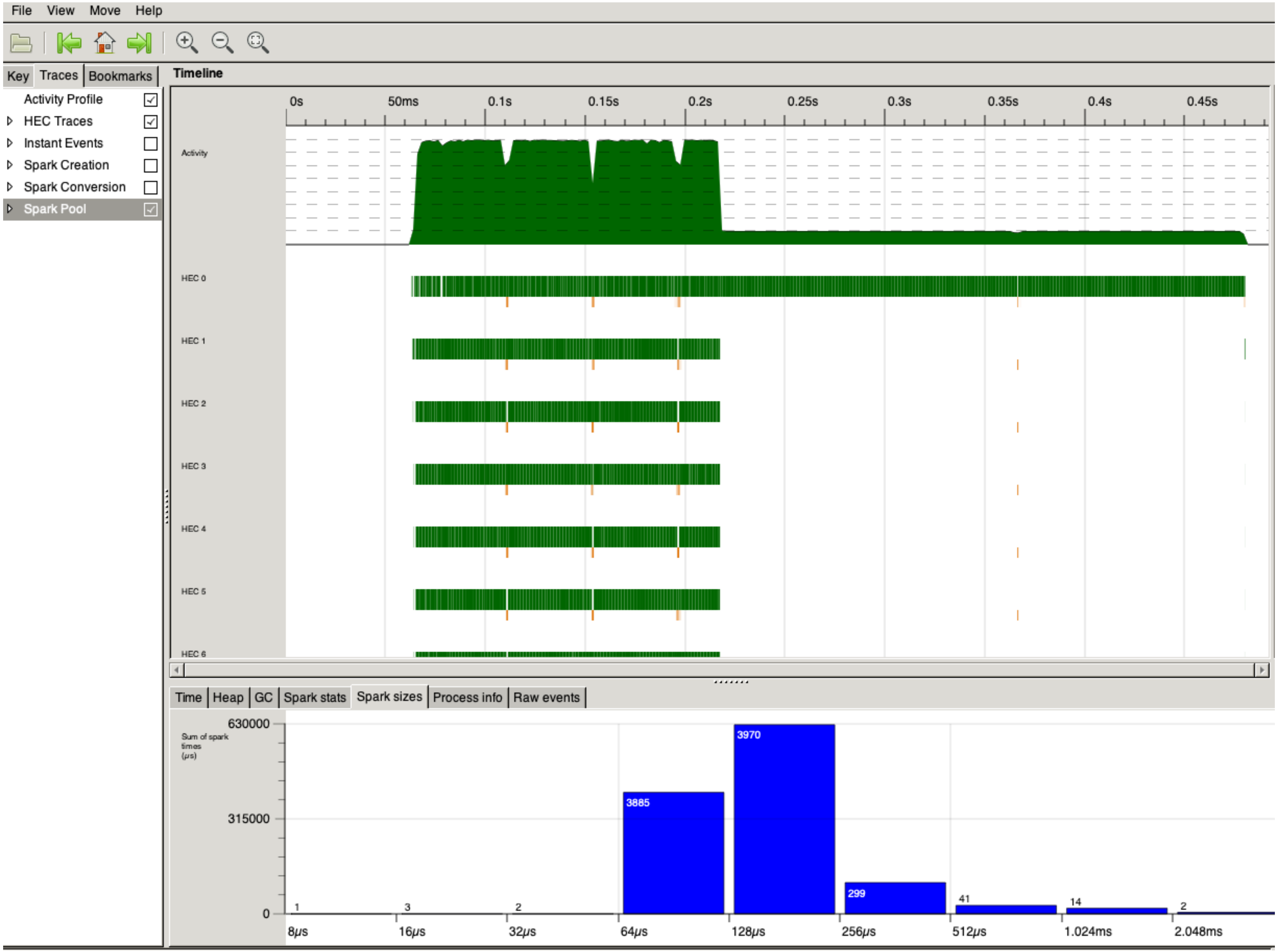
```
INIT    time    0.003s  (  0.009s elapsed)
MUT     time    1.346s  (  0.410s elapsed)
GC      time    0.010s  (  0.003s elapsed)
EXIT    time    0.001s  (  0.000s elapsed)
Total   time    1.361s  (  0.423s elapsed)
```



Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	10000	8204	1796	0	0	0
HEC 0	0	2721	0	0	0	0
HEC 1	10000	267	1796	0	0	0
HEC 2	0	2613	0	0	0	0
HEC 3	0	2603	0	0	0	0

ture1/l2/l2-exe.eventlog (21005 events, 0.445s)


File View Move Help
Key Traces Bookmarks
Activity Profile
HEC Traces
Instant Events
Spark Creation
Spark Conversion
Spark Pool
Timeline
0s 50ms 0.1s 0.15s 0.2s 0.25s 0.3s 0.35s 0.4s 0.45s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
HEC 6
Time Heap GC Spark stats Spark sizes Process info Raw events
630000
315000
0
1 3 2 3885 3970 299 41 14 2
8μs 16μs 32μs 64μs 128μs 256μs 512μs 1.024ms 2.048ms

converted	real parallelism at runtime
overflowed	no room in spark pool
dud	first arg of rpar already eval'ed
GC'd	sparked expression unused (removed from spark pool)
fizzled	uneval'd when sparked, later eval'd independently => removed

our parallel map

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

parallel map

+ Captures a pattern of parallelism
+ good to do this for standard higher order function like map
+ can easily do this for other standard sequential patterns

```
return (b:bs)
```

BUT

- had to write a new version of map
- mixes algorithm and dynamic behaviour

```
return (b:bs)
```

Evaluation Strategies

Raise level of abstraction

Encapsulate parallel programming idioms as reusable components that can be composed

Strategy (as of 2010)

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

function

evaluates its input to some degree

traverses its argument and uses rpar and rseq to express dynamic behaviour / sparking

returns an equivalent value in the Eval monad

using

```
using :: a -> Strategy a -> a

x `using` strat = runEval (strat x)
```

Program typically applies the strategy to a structure and then uses the returned value, discarding the original one (which is why the value had better be equivalent)

An almost identity function that does some evaluation and expresses how that can be parallelised

“The key idea in our reformulation is that a strategy returns a new version of its argument, in which the sparked computations have been embedded.”

Basic strategies

```
r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

Basic strategies

```
r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

NO evaluation

Basic strategies

```
r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

spark x

Basic strategies

```
r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

evaluate x
to WHNF

Basic strategies

```
r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a =>
rdeepseq x = rnf x `pseq` return x
```

fully evaluate x

evalList

```
evalList :: Strategy a -> Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

evalList

```
evalList :: Strategy a -> Strategy [a]
evalList s []          = return []
evalList s (x:         = do x' <- s x
                            xs' <- evalList s xs
```

Takes a Strategy on a and returns a Strategy on lists of a
Building strategies from smaller ones

parList

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Composing strategies

```
dot :: Strategy a -> Strategy a -> Strategy a
strat2 `dot` strat1 = strat2 . runEval . strat1
```

In reality

```haskell
evalList :: Strategy a -> Strategy [a]
evalList = evalTraversable
```

```haskell
parList :: Strategy a -> Strategy [a]
parList = parTraversable
```

In reality

```
evalList :: Strategy a -> Strategy [a]
evalList = evalTraversable
```

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

The equivalent of evalList and of parList are available for many data structures (Traversable). So defining parX for many X is really easy

=> generic strategies for data-oriented parallelism

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 _|_)

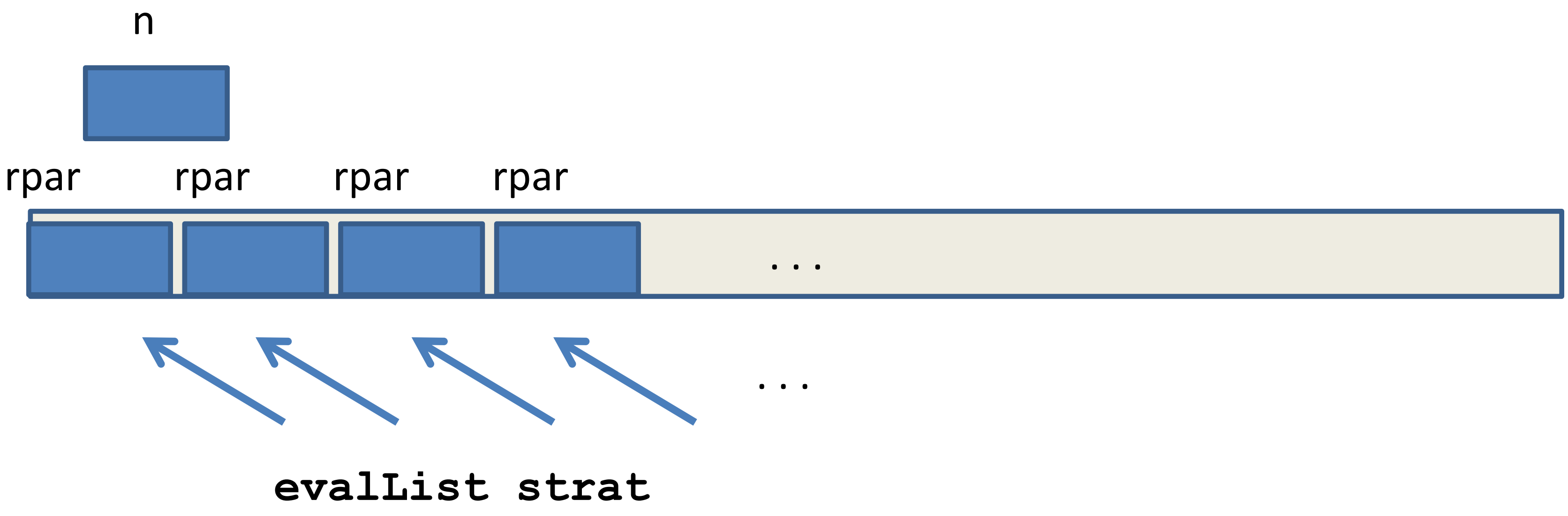
```
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat xs
  | n <= 1    = parList strat xs
  | otherwise
    = concat `fmap`  parList (evalList strat)(chunk n xs)
```

```haskell
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat xs
  | n <= 1    = parList strat xs
  | otherwise
    = concat `fmap`  parList (evalList strat)(chunk n xs)
```

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

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

```
parListChunk n strat
```



```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $ runEval $ pMap foo (reverse [1..10000])
```

Now

```
print $ sum $
(map foo (reverse [1..10000]) `using` parListChunk 50 rdeepseq )
```

Question: how many sparks? Why?

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $ runEval $ pMap foo (reverse [1..10000])
```

Now

```
print $ sum $
 (map foo (reverse [1..10000]) `using` parListChunk 50 rdeepseq )
```

SPARKS: 200 (200 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $
```

Remember not to be a control freak, though. Generating plenty of sparks gives the runtime the freedom it needs to make good choices (=> Dynamic partitioning for free)

Now

```
print $ sum $
(map foo (reverse [1.   `using` parListChunk 50 rdeepseq )
```

SPARKS: 200 (200 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

Research questions (surprisingly unexplored)

What is a good set of combinators for expressing and controlling parallelism, including granularity?

Can recent(ish) developments in the Haskell type system (such as type level natural numbers) help??

Divide and conquer

Capturing patterns of parallel computation is a major strong point of strategies

D&C is a typical example (see also parBuffer, parallel pipelines etc)

These are covered in Lab A

Skeletons

- encode fixed set of common coordination patterns and provide efficient parallel implementations (Cole, 1989)

- Popular in both functional and non-functional languages. See particularly Eden (Loogen et al, 2005)

A difference: one can / should roll ones own strategies

Strategies: summary

+ elegant redesign by Marlow et al   (Haskell 10)

+ better separation of concerns

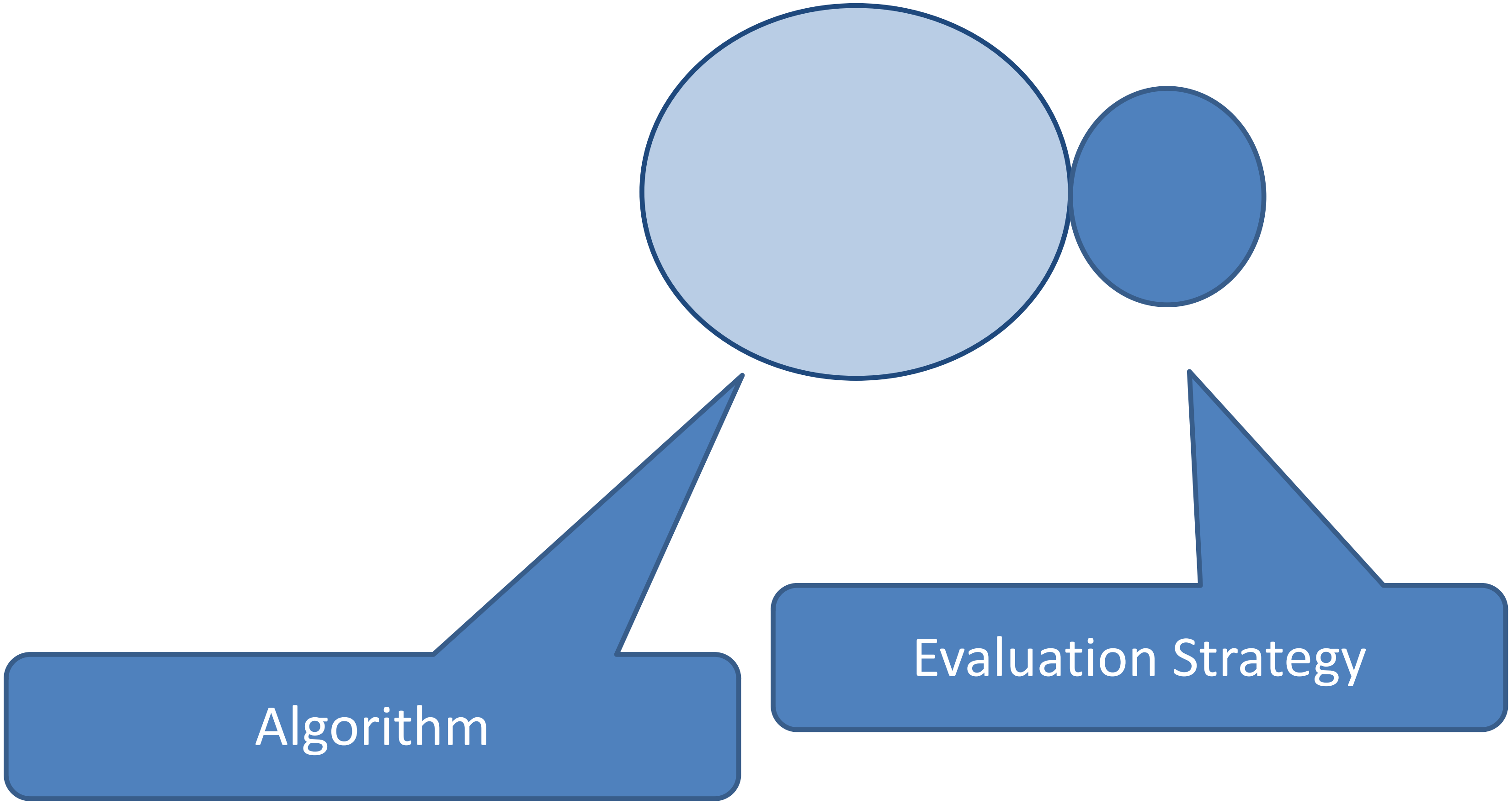
+ Laziness is essential for modularity

+ generic strategies for (Traversable) data structures

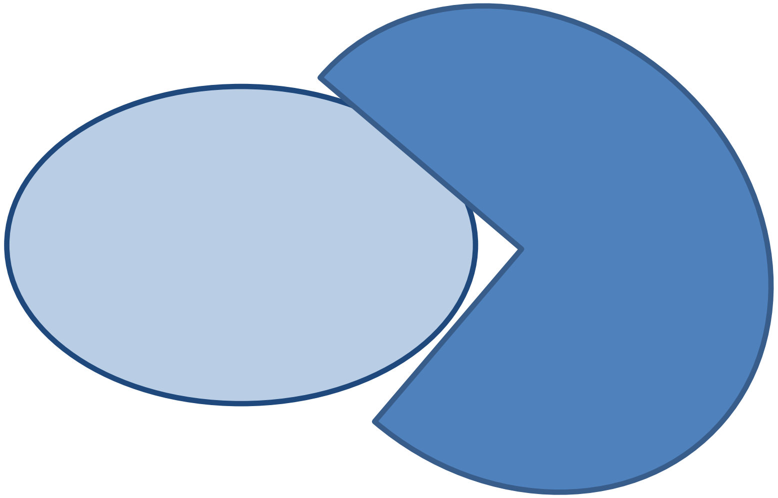
+ Marlow's book contains a nice kmeans example. Read it!

- Having to think so much about evaluation order is worrying! Laziness is not only good here. (Cue the Par Monad Lecture!)

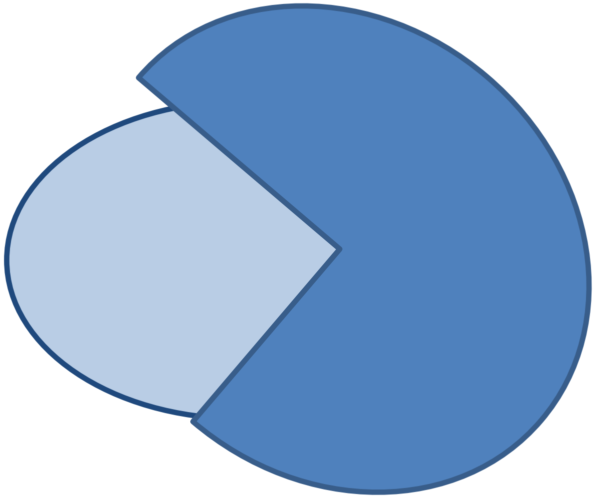
Strategies: summary



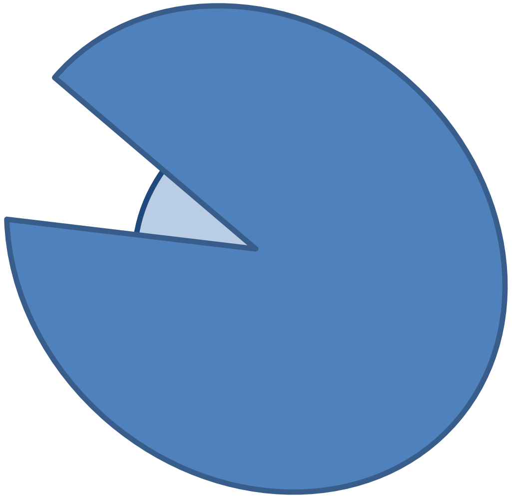
Better visualisation

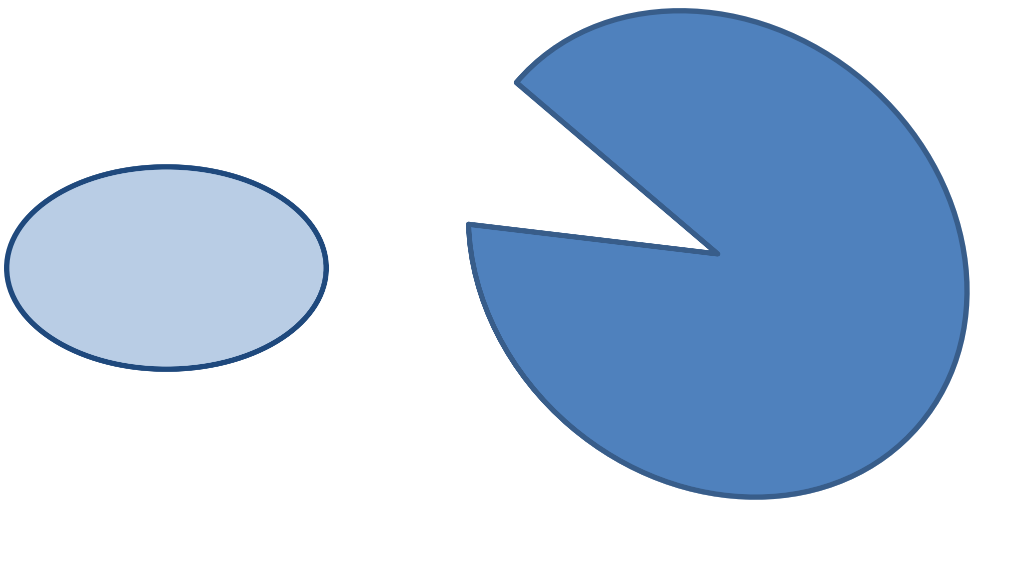


Better visualisation



Better visualisation





Landscape for parallel Haskell (Marlow)

Parallel

par / pseq (1) (monads (2))

Strategies (3)

Par Monad. (4)

DPH, Repa , Accelerate (Keller guest lec.)

Landscape for parallel Haskell (Marlow)

Parallel

- par / pseq (1) (monads (2))
- Strategies (3)
- Par Monad. (4)
- Repa, DPH, Accelerate (Keller guest lec.)

Concurrent

- ForkIO
- MVAR
- STM
- async
- Cloud Haskell

parallel FP

Parallel

- par / pseq (1) (monads (2))
- Strategies (3)
- Par Monad. (4)
- Repa, DPH, Accelerate (Keller guest lec.)
- Parallelising QuickCheck in Haskell
- Data Parallel Programming (NESL) (5)
  - Futhark Java

Parallel/ Concurrent

- robust parallel programming in Erlang
- Map Reduce
- noSQL databases etc.
- (STM)