

# Parallel Functional Programming

## Lecture 2

Mary Sheeran

(with thanks to Simon Marlow for use of slides)

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

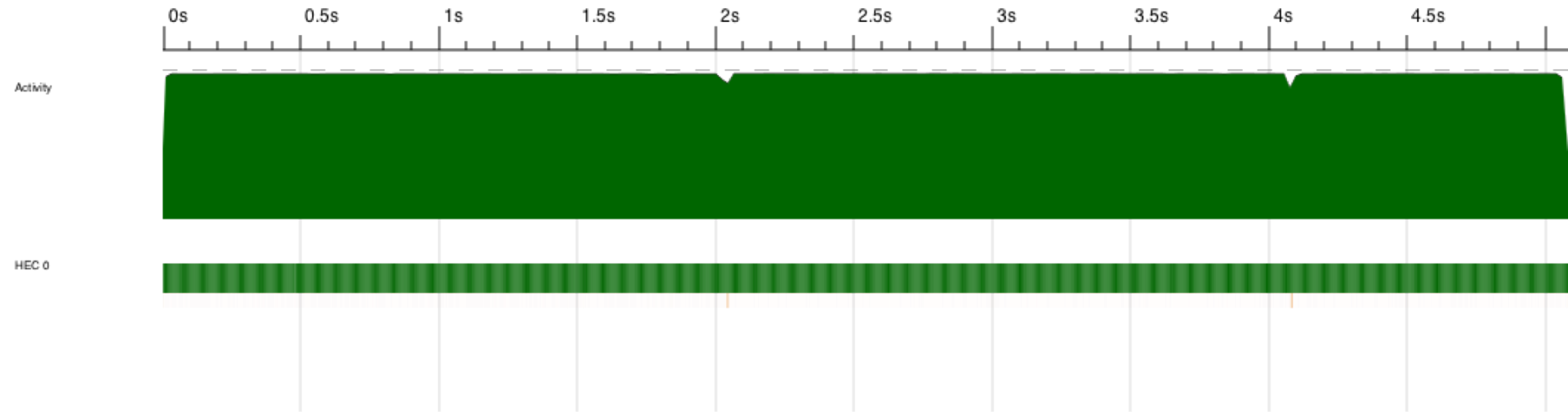
# Remember nfib

```
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

# Sequential



nfib 40

# Explicit 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

# Explicit Parallelism

x ` par ` y

# Explicit sequencing

`pseq x y`

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

# Explicit 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)
```



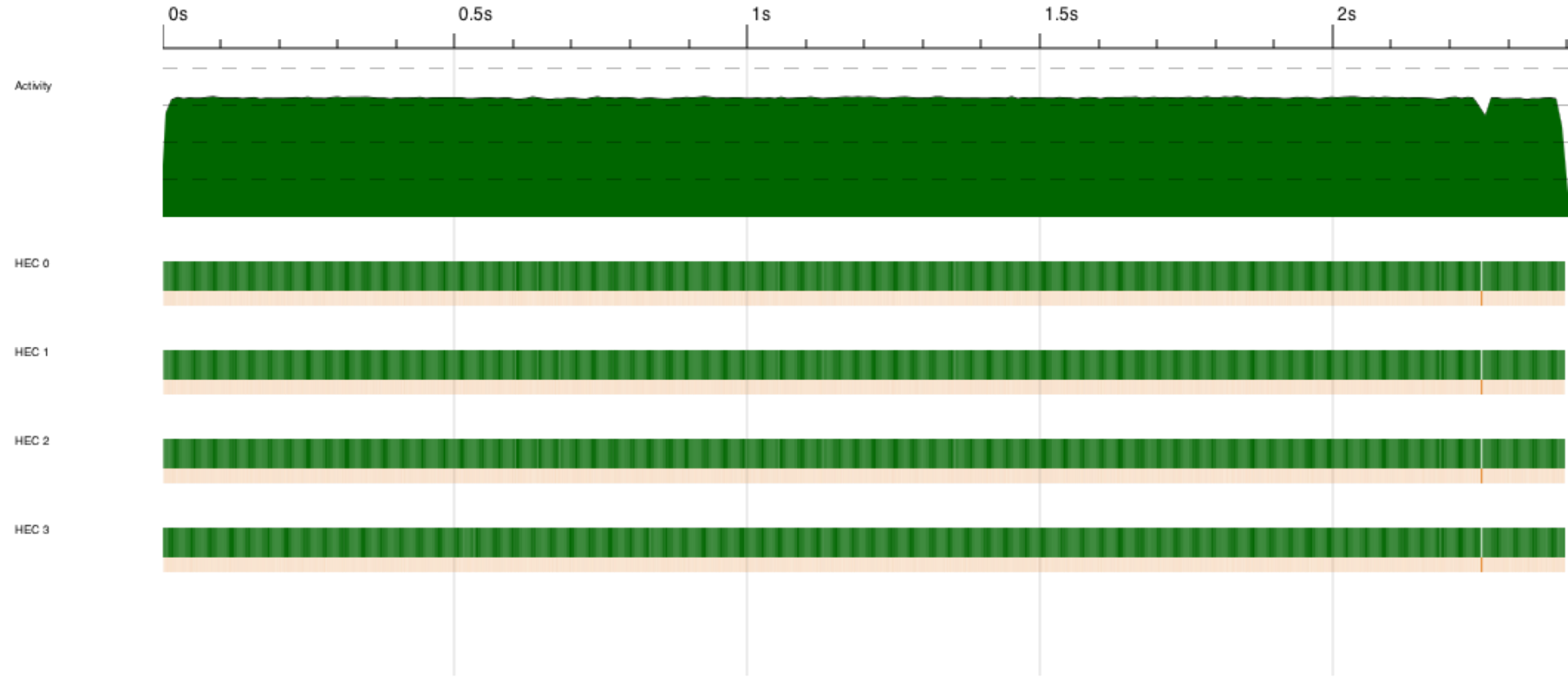
# Using par and pseq

```
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 ...*)

# Looks promsing



# ensing

0s

1.5s

2s

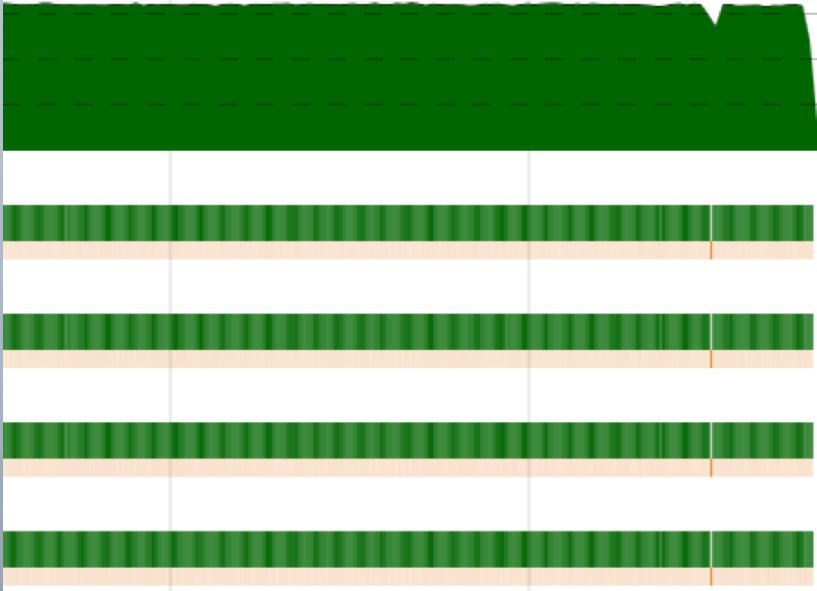
Activity

HEC 0

HEC 1

HEC 2

HEC 3



# What's happening?

```
$ ./NF +RTS -N4 -s
```

-s to get stats

# Hah

331160281

...

SPARKS: 165633686 (105 converted, 0 overflowed, 0 dud, 165098698 GC'd, 534883 fizzled)

INIT	time	0.00s	(	0.00s	elapsed)
MUT	time	2.31s	(	1.98s	elapsed)
GC	time	7.58s	(	0.51s	elapsed)
EXIT	time	0.00s	(	0.00s	elapsed)
Total	time	9.89s	(	2.49s	elapsed)

# Hah

331160281

...

SPARKS: 165633686 (105 converted, 0 overflowed, 0 dud, 165098698 GC'd, 534883 fizzled)

INIT	time	0.00s	( 0.00s elapsed)
MUT	time	2.31s	( 2.31s elapsed)
GC	time	7.58s	( 0.50s elapsed)
EXIT	time	0.00s	( 0.00s elapsed)
Total	time	9.89s	( 2.49s 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 (13 converted, 0 overflowed, 0 dud, 0 GC'd, 75 fizzled)

INIT	time	0.00s	( 0.01s elapsed)
MUT	time	2.42s	( 1.36s elapsed)
GC	time	3.04s	( 0.04s elapsed)
EXIT	time	0.00s	( 0.00s elapsed)
Total	time	5.47s	( 1.41s elapsed)



# 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

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

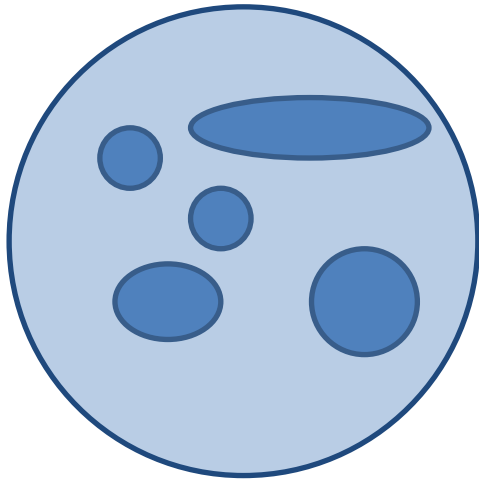
Pass an unevaluated computation to par

It must be somewhat expensive

Make sure the result is not needed for a bit

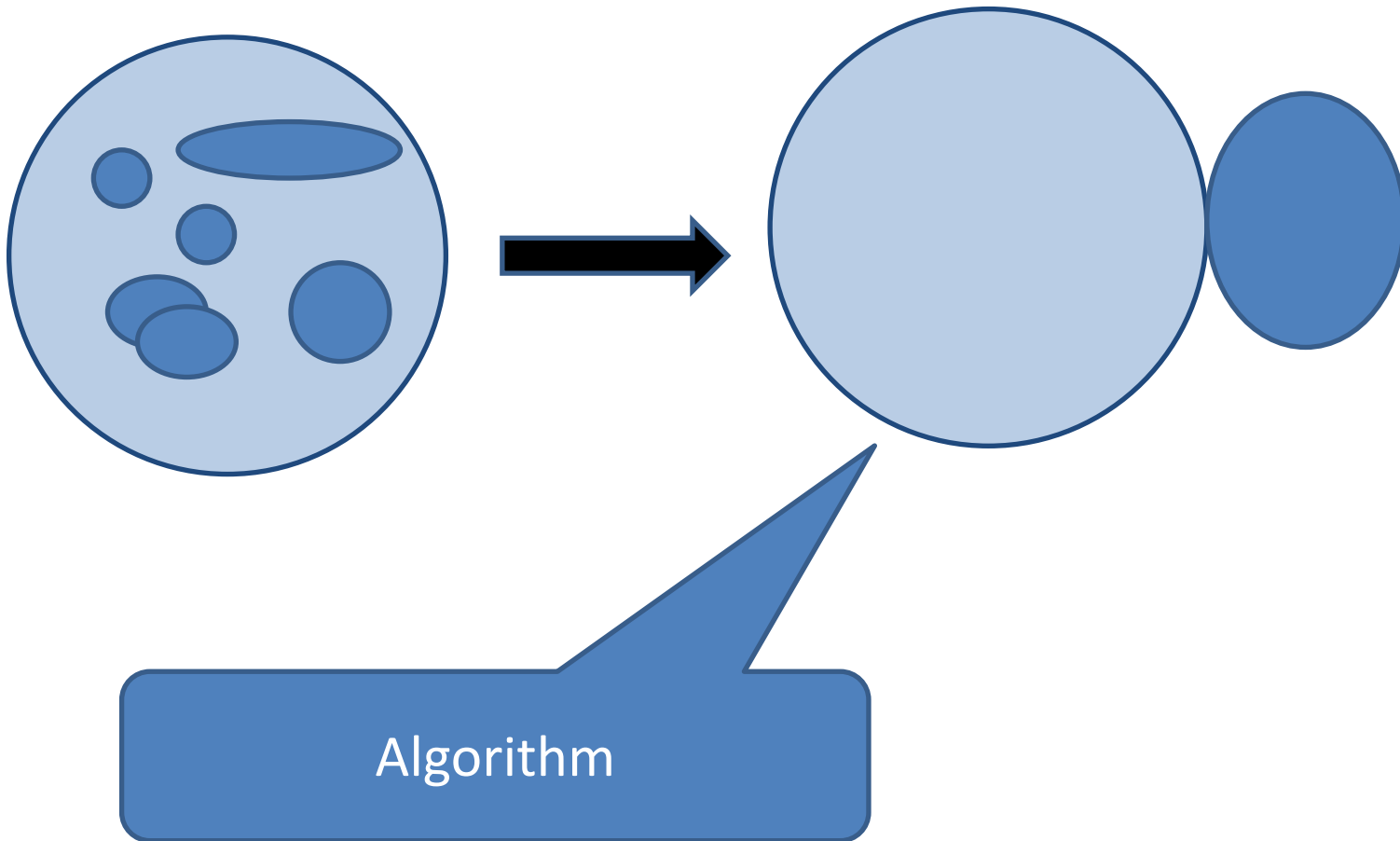
Make sure the result is shared by the rest of the program

# Even if you get it right

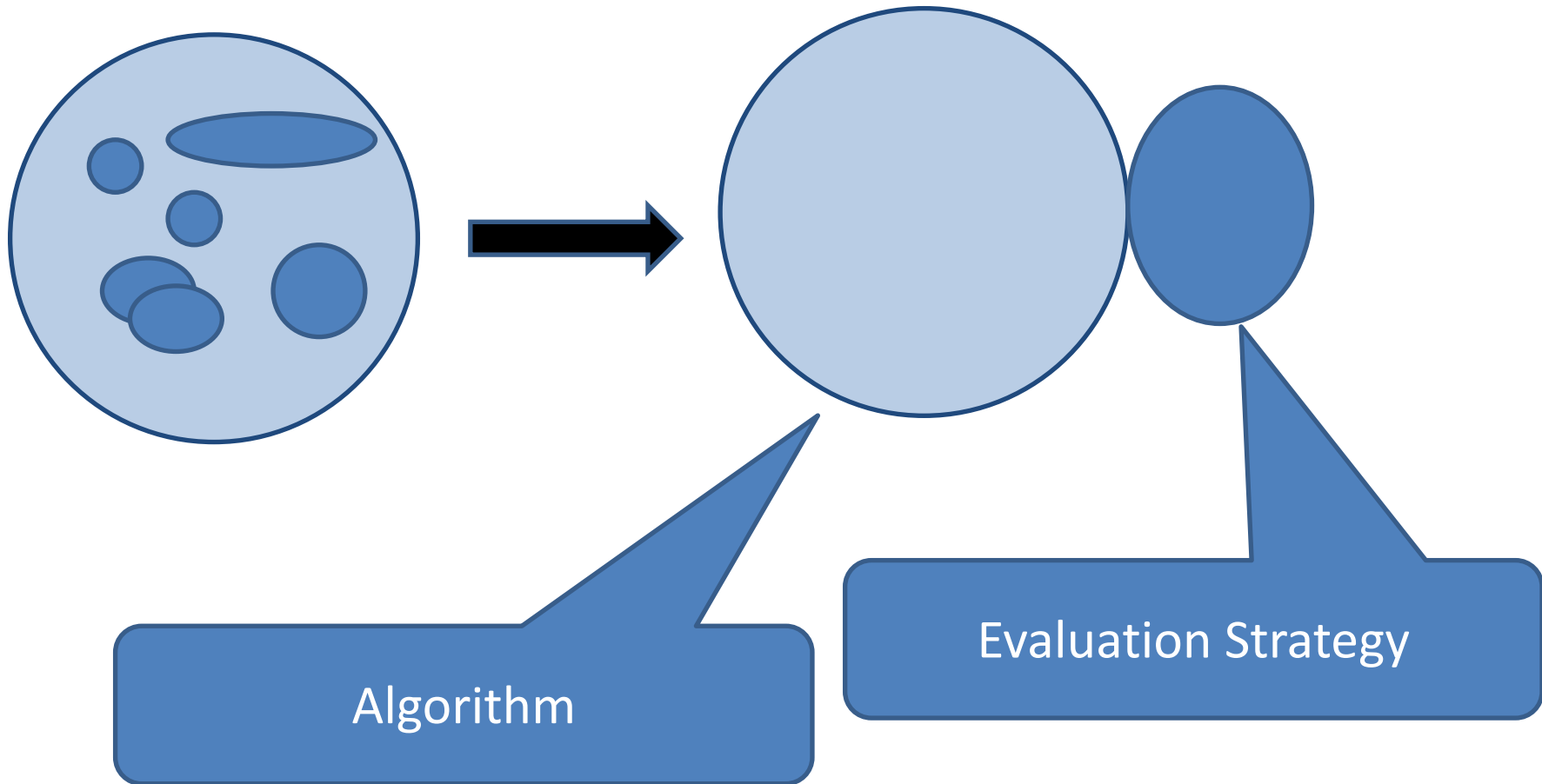


Original code + par + pseq + rnf etc.  
can be opaque

# 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

## *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 <sup>†</sup>

*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 <sup>†</sup>

*Department of Computing Science, University of Glasgow, Glasgow, UK*

JFP 1998



359

## **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 <sup>†</sup>

*Department of Computing Science, University of Glasgow, Glasgow, UK*

JFP 1998



359

## **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



88

# Papers

*Algorithm + Strategy = Parallelism*

P.W. TRINDER

*Department of Computing Science, University of Glasgow*

K. HAMMOND

*Division of Computing Science, University of St Andrews*

H.-W. LOIDL AND S.L. PEYTON JONES

*Department of Computing Science, University of Glasgow*

Redesigns strategies

richer set of parallelism combinators

Better specs (evaluation order)

Allows new forms of coordination

generic regular strategies over data structures

speculative parallelism

monads everywhere 😊

Presentation is about New Strategies

**Seq no more: Better Parallelism**

Simon Marlow

Microsoft Research, Cambridge, UK  
simonmar@microsoft.com

Patrick Maier

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

Heriot-Watt University, Edinburgh, UK

H.W.Loid@hw.ac.uk

Mustafa K. Awad

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

Phil Trinder

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

FRASER 10

# The Eval monad

```
import Control.Parallel.Strategies

data Eval a
instance Monad Eval

runEval :: Eval a -> a

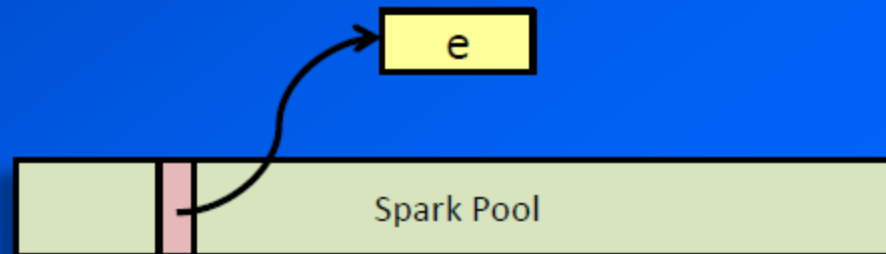
rpar :: a -> Eval a
rseq :: a -> Eval a
```

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

# What does *rpar* actually do?

```
x <- rpar e
```

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



# 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 (n-1))
    nf2 <- rseq (qfib (n-1))
    return (nf1 + nf2 + 1)
```

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 (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```

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

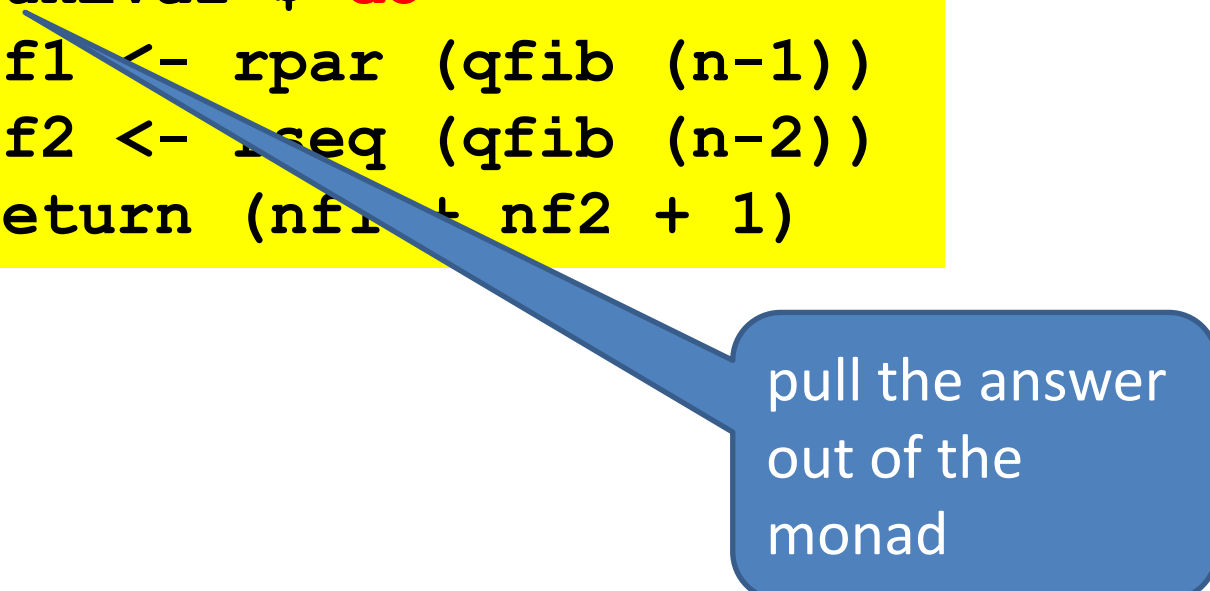
```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```



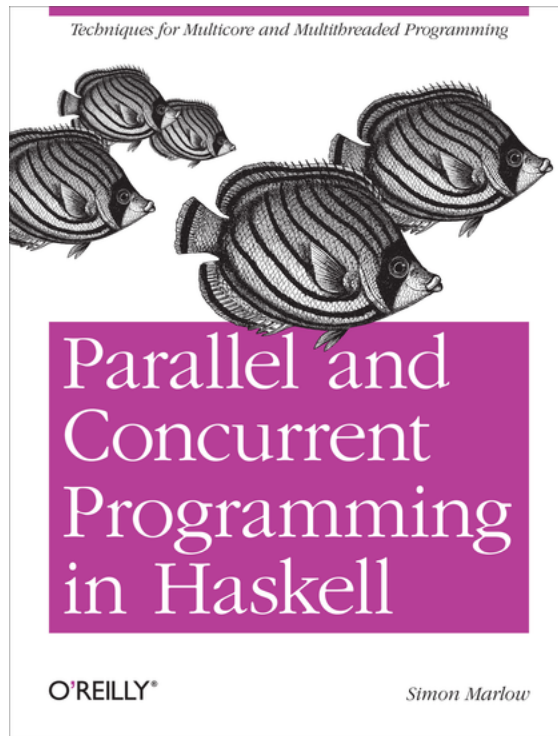
the result

# 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



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])
```



# compile

```
ghc -O2 -threaded -rtsopts L1.hs
```

# run & get stats

```
$ ./L1 +RTS -N4 -s -A100M
```

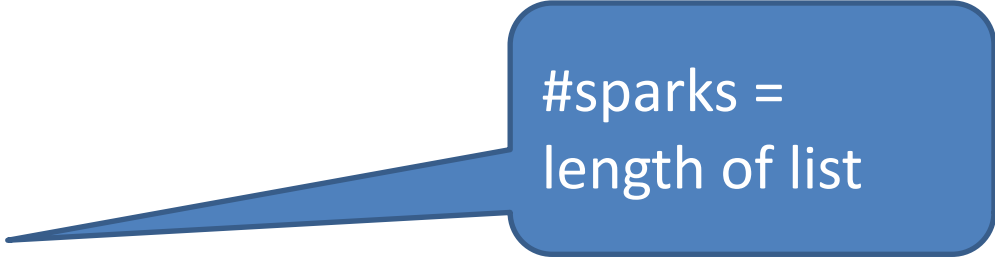
# run & get stats

```
$ ./L1 +RTS -N4 -s -A100M
```

Sets GC nursery size  
Effectively turns off the collector and  
removes its effects from benchmarking  
(See notes in Lab A)

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)

# Compile for Threadscope

```
ghc -O2 -threaded -rtsopts -eventlog L1.hs
```

Using prebuilt binaries for Threadscope is the way to go:

<https://www.stackage.org/package/threadscope>

# Run for Threadscope

```
$ ./L1 +RTS -N4 -lf -A100M
```





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

# withStrategy

```
withStrategy :: Strategy a -> a -> a  
withStrategy = flip using
```



# Composing strategies

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

# Composing strategies

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

==

```
strat2 . withStrategy strat1
```

# 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
```

NO evaluation

```
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
```



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 => Strategy 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:xs) = 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)
```

# In reality

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

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

# In reality

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

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

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”:
  - $x \text{ `using` } s \approx x$
  - 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)
```

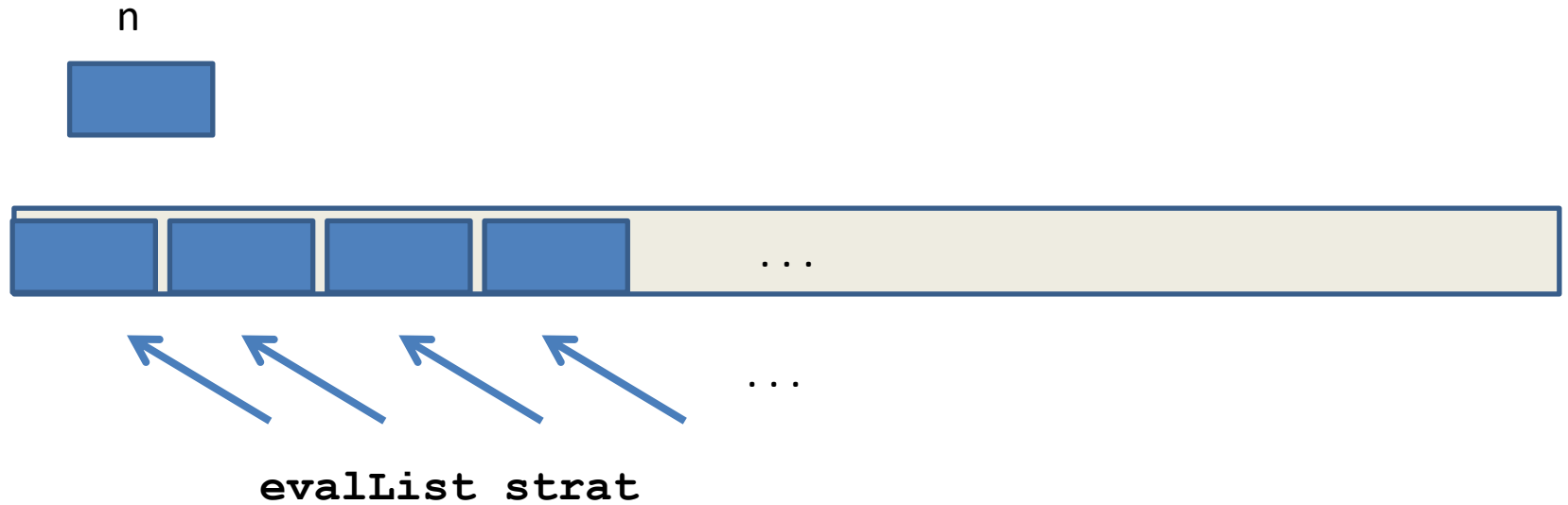
```
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat xs
  | n <= 1      = parList strat xs
  | otherwise
    = concat `fmap` parList (evalList strat) (chunk n xs)
```

```
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 )
```

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

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

Before

```
print $ sum $
```

Now

```
print $ sum $  
(map foo (reverse [1..1000000]) using `parListChunk 50 rdeepseq` )
```

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)

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

```
import Criterion.Main
```

```
check k = sum $  
  (map foo (reverse [1..10000])  
   `using`  
   parListChunk k rdeepseq )
```

```
main = defaultMain [bench "L1" (nf check 100)]
```

```
$ ./L1 +RTS -N4 -A100M
```

```
benchmarking L1
```

```
time                510.2  $\mu$ s    (503.5  $\mu$ s .. 517.3  $\mu$ s)
```

```
                   0.998 R2    (0.997 R2 .. 0.999 R2)
```

```
mean                512.4  $\mu$ s    (508.1  $\mu$ s .. 518.3  $\mu$ s)
```

```
std dev             18.19  $\mu$ s    (14.85  $\mu$ s .. 23.18  $\mu$ s)
```

```
variance introduced by outliers: 28% (moderately inflated)
```

# using is not always what we need

- Trying to pull apart algorithm and coordination in qfib (from earlier) doesn't really give a satisfactory answer (see Haskell 10 paper)

(If the worst comes to the worst, one can get explicit control of threads etc. in concurrent Haskell, but determinism is lost... )

# 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.)

```
divConq :: (a -> b)
         -> a
         -> (a -> Bool)
         -> (b -> b -> b)
         -> (a -> Maybe (a, a))
         -> b
```

function on base cases  
input  
par threshold reached?  
combine  
divide  
result

# Divide and Conquer

```
divConq f arg threshold conquer divide = go arg
  where
    go arg =
      case divide arg of
        Nothing      -> f arg
        Just (l0,r0) -> conquer l1 r1 'using' strat
          where
            l1 = go l0
            r1 = go r0
            strat x = do r l1; r r1; return x
                  where r | threshold arg = rseq
                          | otherwise    = rpar
```

Separates algorithm and strategy

A first inkling that one can probably do interesting things by programming with strategies



# Skeletons

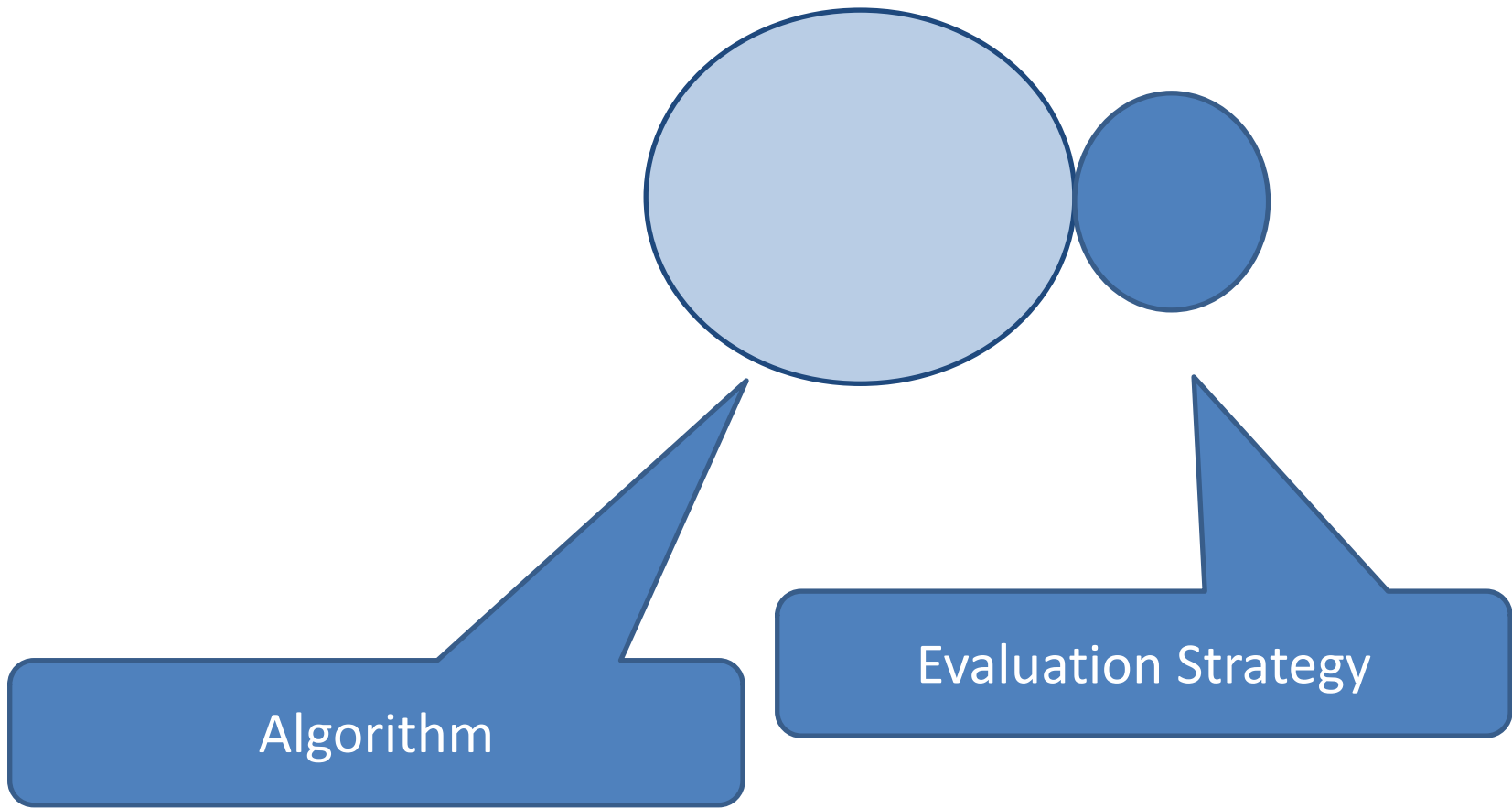
- encode fixed set of common coordination patterns and provide efficient parallel implementations (Cole, 1989)
- Popular in both functional and non-functional languages. See particularly Eden (Loogen et al, 2005)

A difference: one can / should roll ones own strategies

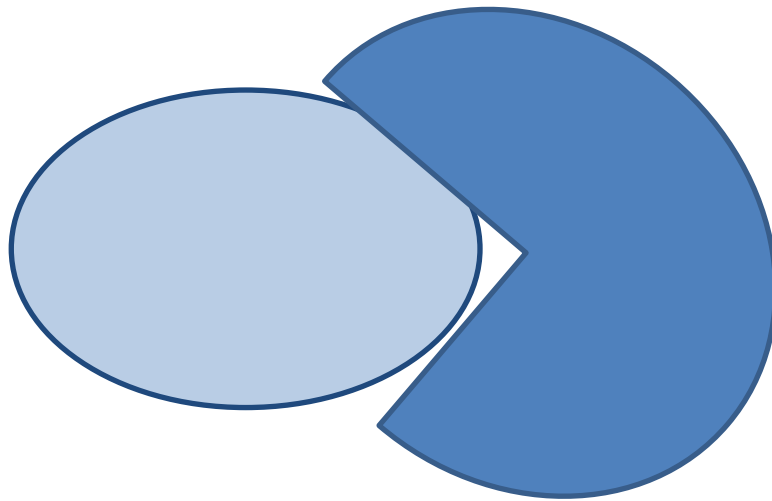
# Strategies: summary

- + 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 contain 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!)**

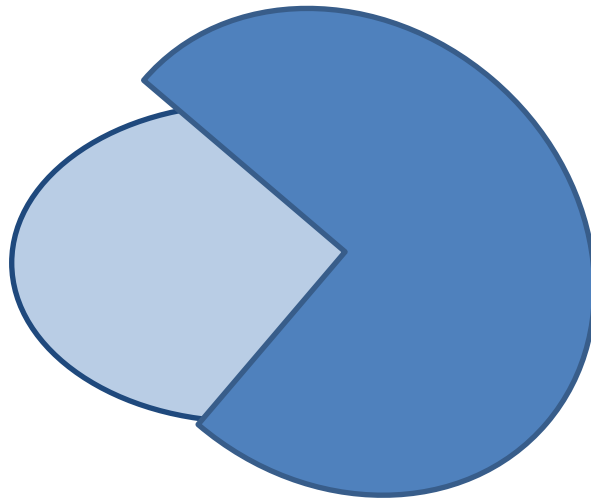
# Strategies: summary



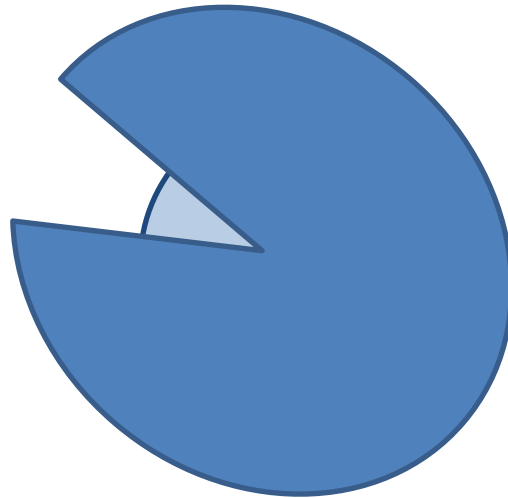
# Better visualisation

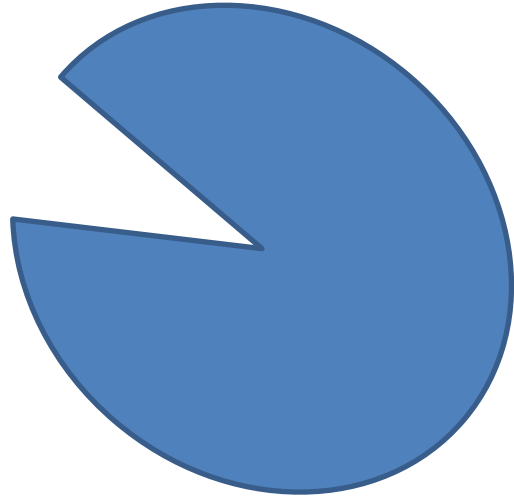
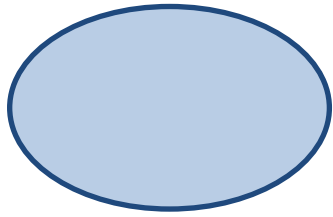


# Better visualisation



# Better visualisation





# Simon Marlow's landscape for parallel Haskell

- Parallel
  - par/pseq 1
  - Strategies 2
  - Par Monad 3
  - Repa
  - Accelerate 4
  - DPH
- Concurrent
  - forkIO
  - MVar
  - STM
  - async
  - Cloud Haskell

Haxl?



Course reps??

# In the meantime

Read papers and PCPH

Start on Lab A (due 23.59 April 12)

Exercise class tomorrow at 15.15 (EC)

Note office hours of TAs

Markus, tues 10.00-11.00

Max, thu 14.00-15.00

**Use them!**