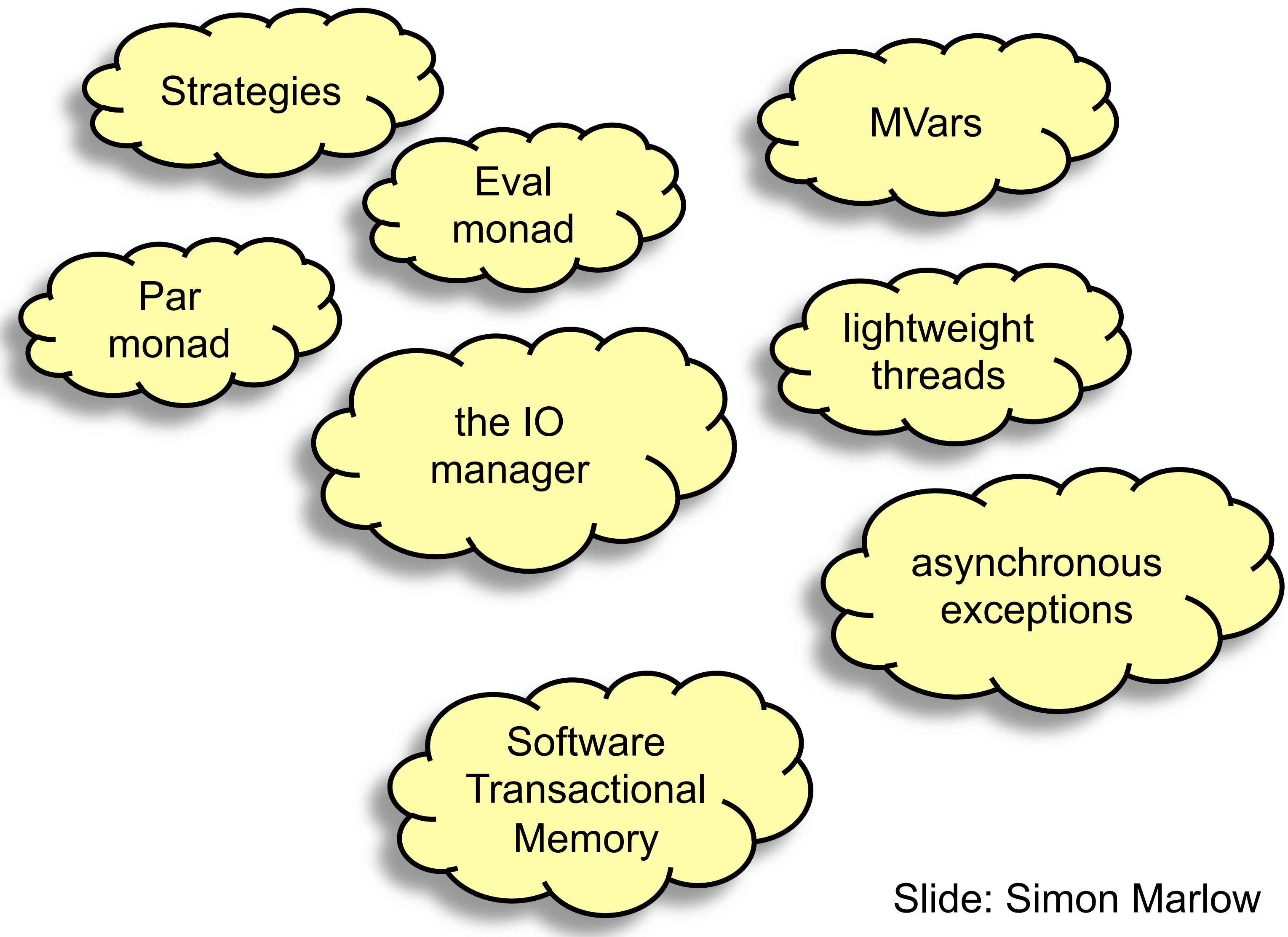
Parallel Programming

David Sands

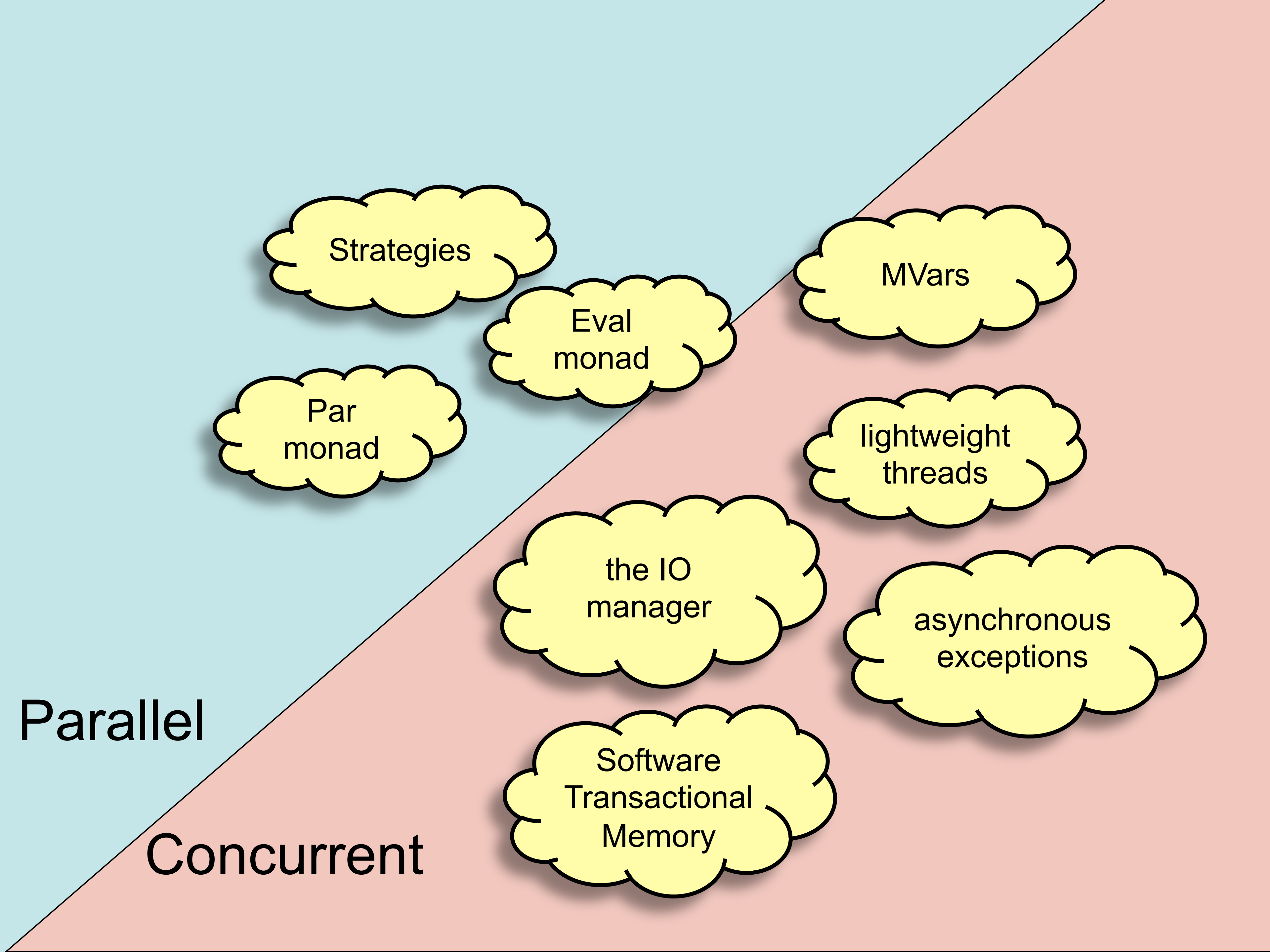
Slides borrowed and adapted from Simon Marlow's page

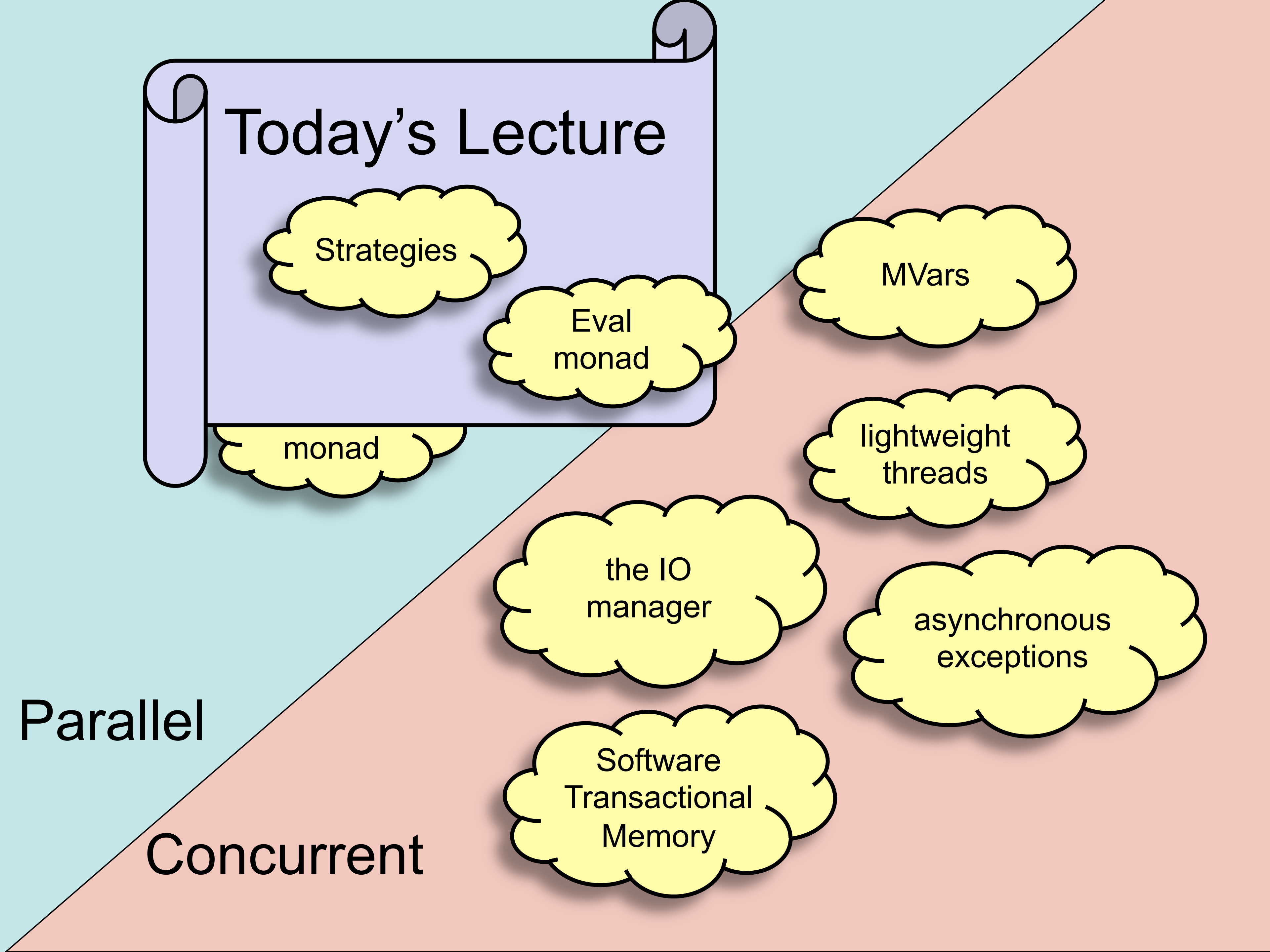
http://community.haskell.org/~simonmar/CEFP1.pdf

Parallel and Concurrent Haskell ecosystem

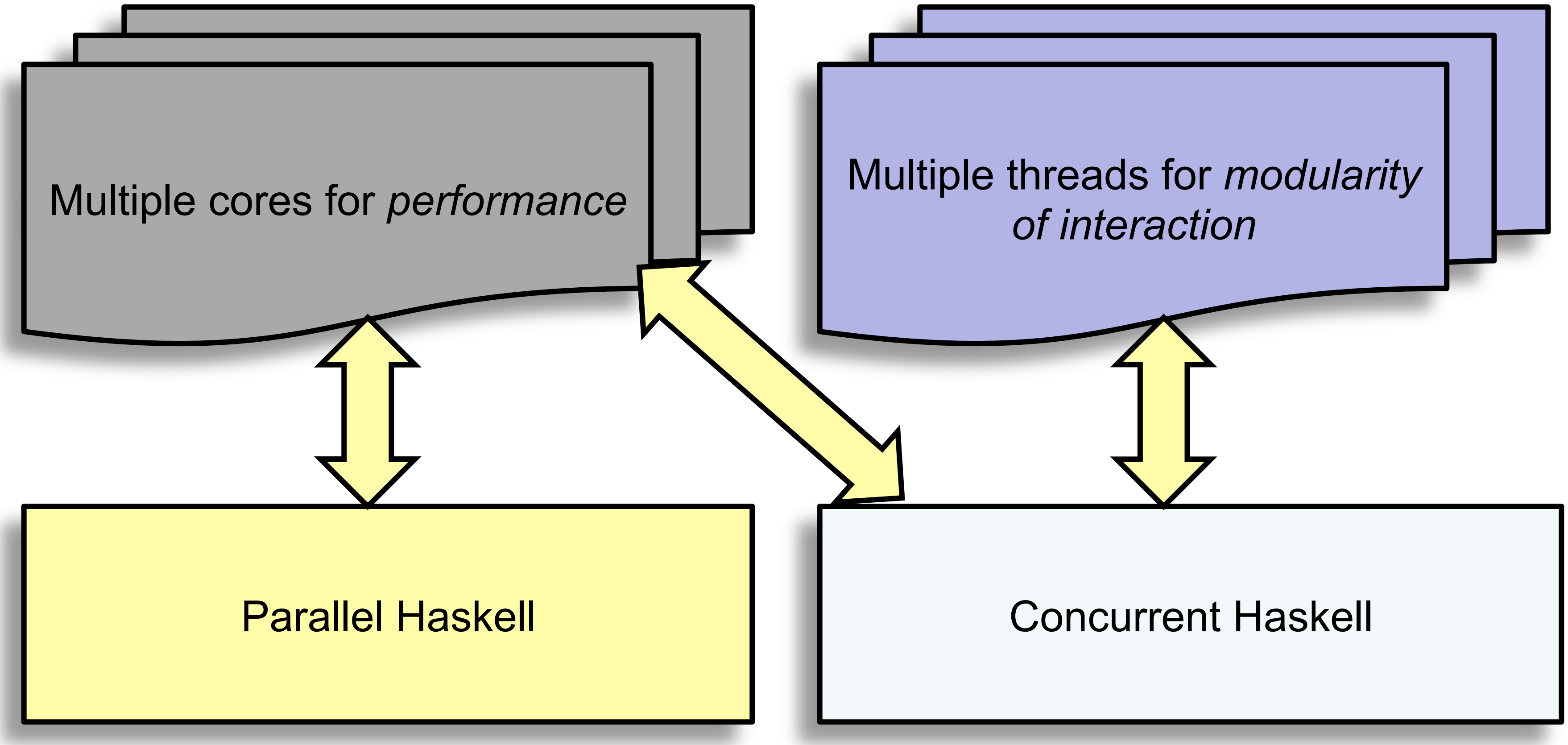


Slide: Simon Marlow


Strategies
Eval monad
Par monad
MVars
lightweight threads
the IO manager
asynchronous exceptions
Software Transactional Memory
Parallel
Concurrent


Today's Lecture
Strategies
Eval monad
monad
MVars
lightweight threads
the IO manager
asynchronous exceptions
Software Transactional Memory
Parallel
Concurrent

Parallelism vs. Concurrency



Slide: S. Marlow

Parallelism vs. Concurrency

- Primary distinguishing feature of Parallel Haskell: determinism
  - The program does “the same thing” regardless of how many cores are used to run it.
  - No race conditions or deadlocks
  - add parallelism without sacrificing correctness
  - Parallelism is used to speed up pure (non-IO monad) Haskell code

Slide: S. Marlow

Parallelism vs. Concurrency

- Primary distinguishing feature of Concurrent Haskell: threads of control
  - Concurrent programming is done in the IO monad
    - because threads have *effects*
    - effects from multiple threads are interleaved nondeterministically at runtime.
  - Concurrent programming allows programs that interact with multiple external agents to be *modular*
    - the interaction with each agent is programmed separately
    - Allows programs to be structured *as* a collection of interacting agents (actors)

Slide: S. Marlow

Parallel Haskell

- Basic primitives: par and pseq
- parallelise use of Sudoku solver
- use ThreadScope to profile parallel execution

- do dynamic rather than static partitioning
- measure parallel speedup
  - use Amdahl's law to calculate possible speedup

– Evaluation Strategies
- build simple Strategies

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
```

Slide: S. Marlow

Solving Sudoku problems

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

```
import Sudoku

f = "sudoku17.1000.txt”

main :: IO ()
main = do
    grids <- fmap lines $ readFile f
    let solutions = map solve grids
    print $ all isJust solutions
```

Compile

- Optimisation –O2
- Runtime options

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$
```

Controlling Evaluation for Parallelism

- *In theory* a compiler should be able to automatically compile pure functional programs to use multiple cores
  - purity ⇒ computations can be freely reordered without changing the result
- *In practice* this is hard. We need to give hints as to which strategy to use
  - but no synchronisation/deadlock issues need to be considered!

Run the program...

```
$ ./sudoku1 +RTS -s
   2,392,127,440 bytes allocated in the heap
      36,829,592 bytes copied during GC
         191,168 bytes maximum residency (11 sample(s))
          82,256 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  4570 collections,     0 parallel,  0.14s,  0.13s elapsed
  Generation 1:    11 collections,     0 parallel,  0.00s,  0.00s elapsed

...

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.92s  (  2.92s elapsed)
  GC    time    0.14s  (  0.14s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.06s  (  3.06s elapsed)

...
```

par and pseq

`ghc -threaded` uses a threaded runtime system. To make use of it we need to add some parallelism hints to the code

`Control.Parallel` provides

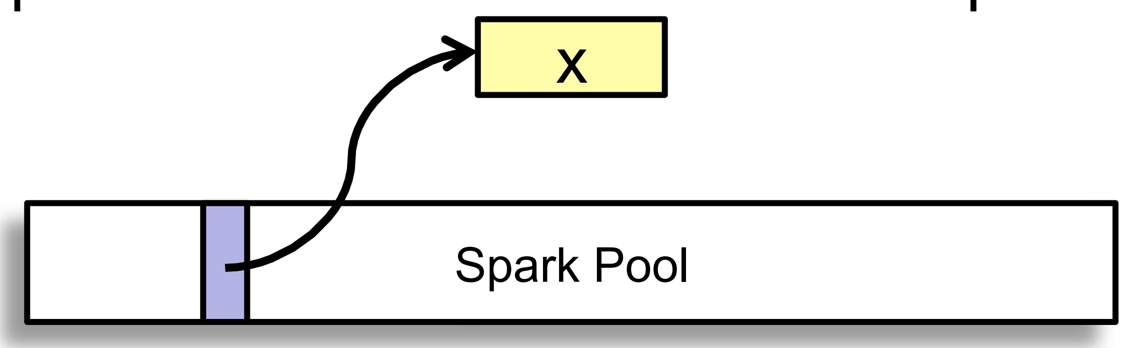
`pseq , par :: a -> b -> b`

- `pseq` – seq but with a *guarantee* of left-to-right evaluation order
- `par` – maybe evaluate left argument (to whnf) possibly in parallel with its right arg.

What does par *actually* do?

```
x `par` y
```

- par creates a *spark* by writing an entry in the *spark pool*
  - par 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!



Parallelising Sudoku

- Let's divide the work in two, so we can solve each half in parallel:

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

- Now we need something like

```
let solutions = as' `par` bs' `pseq` as' ++ bs'
print $ all isJust solutions
```

But this won’t work...

- Like seq, par evaluates its argument to Weak Head Normal Form (WHNF)
  - evaluates as far as the *first constructor*
  - e.g. for a list, we get either [] or (x:xs)
  - e.g. WHNF of `map solve (a:as)` would be

    `solve a : map solve as`
- But we want to evaluate the whole list, and the elements

We need ‘deepseq’

```
import Control.DeepSeq(deepseq)
-- deepseq :: NFData a => a -> b -> b
```

- deepseq fully evaluates a nested data structure (its first arg) and returns it’s second
  - e.g. `a `deepseq` b`

  `a` is fully evaluated, including the elements
- uses overloading: the argument must be an instance of NFData
  - instances for most common types are provided by the library

deep

- We need to use deepseq inside a par

```
(as `deepseq` as) `par` …
```

```
deep a = a `deepseq` a
```

- But things in a par should be variables, and should be used later (otherwise the spark might get discarded!). Thus:

```
let as’ = deep as in
         as’ `par` …
```

Using deep

```
main = do
    grids <- fmap lines $ readFile f
    let (as,bs) = splitAt (length grids `div` 2) grids
        as' = deep $ map solve as
        bs' = deep $ map solve bs
        result = all isJust (as' ++ bs')
    bs' `par` as' `pseq` print result
```

- Why bs’ before as’?
  - worked out a little better
  - need performance measurement…

Let's try it...

- Compile sudoku2
  - (add -threaded -rtsopts)
  - run with `sudoku17.1000.txt +RTS -N2`
- Take note of the Elapsed Time

Runtime results...

```
$ ./sudoku1 +RTS -s -N2
True
   2,400,106,440 bytes allocated in the heap
      48,996,296 bytes copied during GC
       2,615,040 bytes maximum residency (7 sample(s))
         326,584 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  2984 collections,  2983 parallel,  0.65s,  0.12s elapsed
  Generation 1:     7 collections,     7 parallel,  0.02s,  0.02s elapsed

  Parallel GC work balance: 1.49 (6106266 / 4103299, ideal 2)

SPARKS: 1 (1 converted, 0 pruned)

  INIT  time    0.01s  (  0.01s elapsed)
  MUT   time    2.25s  (  1.70s elapsed)
  GC    time    0.68s  (  0.14s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    2.93s  (  1.85s elapsed)
```

One spark was created and one was run by a processor

speedup ~1.5 over the sequential version

Why not 2?

- 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 screw up

- less than 100% utilisation
  - parallelism was not created, or was discarded
  - algorithm not fully parallelised – residual sequential computation
  - uneven work loads
  - poor scheduling
  - communication latency

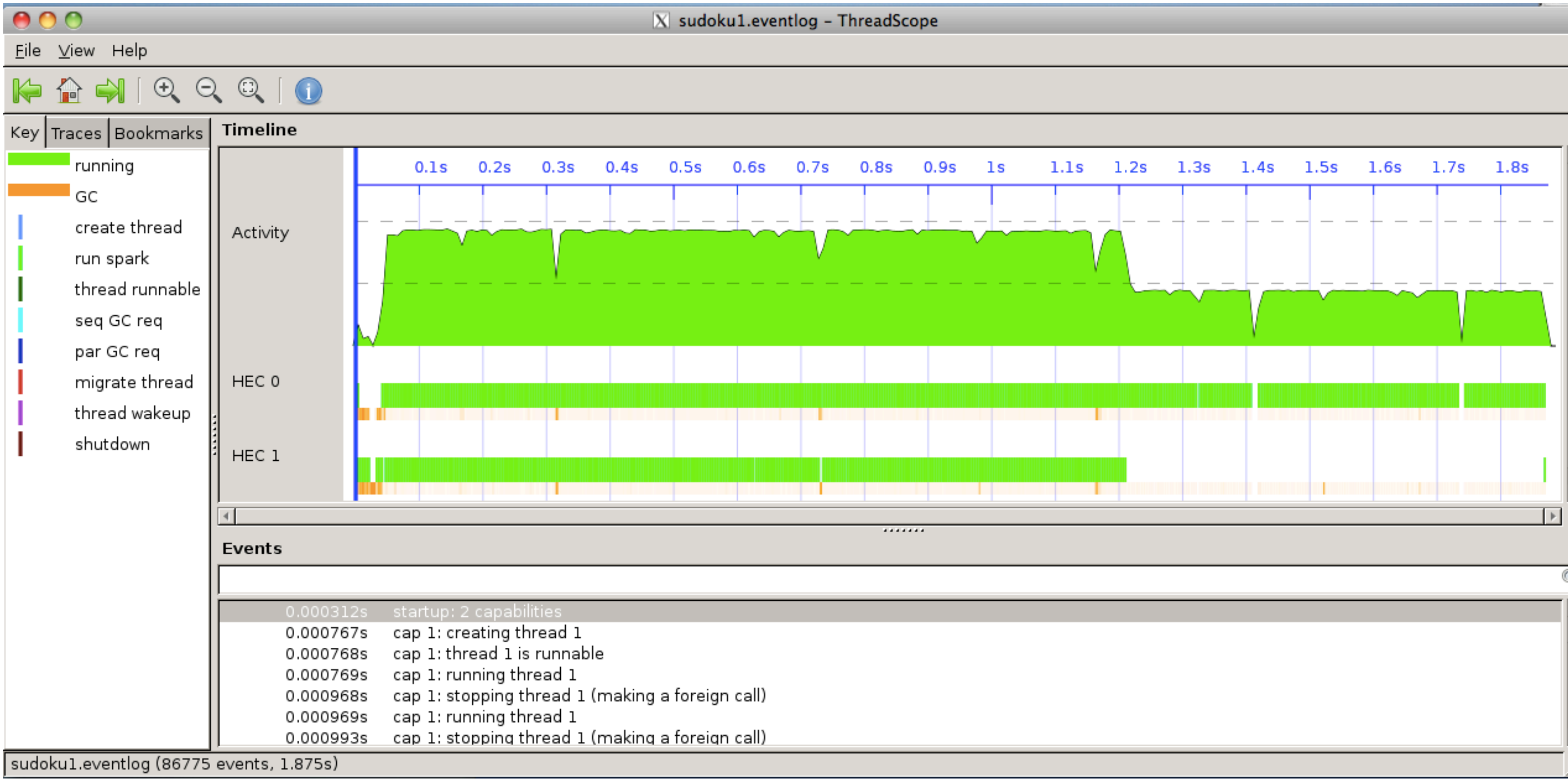
- extra overhead in the parallel version
  - overheads from rpar, work-stealing, deep, ...
  - lack of locality, cache effects...
  - larger memory requirements leads to GC overhead
  - GC synchronisation
  - duplicating work

So we need *tools*

- to tell us why the program isn't performing as well as it could be
- For Parallel Haskell we have ThreadScope

```
$ ghc -O2 sudoku2.hs -threaded -rtsopts -eventlog
$ ./sudoku2 +RTS -N2 -ls
$ threadscope sudoku2.eventlog
```

- -eventlog has very little effect on runtime
  - important for profiling parallelism



sudoku1.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
Activity
HEC 0
HEC 1
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
Events
0.000312s startup: 2 capabilities
0.000767s cap 1: creating thread 1
0.000768s cap 1: thread 1 is runnable
0.000769s cap 1: running thread 1
0.000968s cap 1: stopping thread 1 (making a foreign call)
0.000969s cap 1: running thread 1
0.000993s cap 1: stopping thread 1 (making a foreign call)
sudoku1.eventlog (86775 events, 1.875s)

Uneven workloads...

- So one of the tasks took longer than the other, leading to less than 100% utilisation

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

- One of these lists contains more work than the other, even though they have the same length
  - sudoku solving is not a constant-time task: it is a searching problem, so depends on how quickly the search finds the solution

Partitioning

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

- Dividing up the work along fixed pre-defined boundaries, as we did here, is called *static partitioning*
  - static partitioning is simple, but can lead to under-utilisation if the tasks can vary in size
  - static partitioning does not adapt to varying availability of processors – our solution here can use only 2 processors

Dynamic Partitioning

- GHC’s runtime system provides spark pools to track dynamic work units, and a work-stealing scheduler to assign them to processors
- So all we need to do is use smaller tasks and more pars, and we get dynamic partitioning

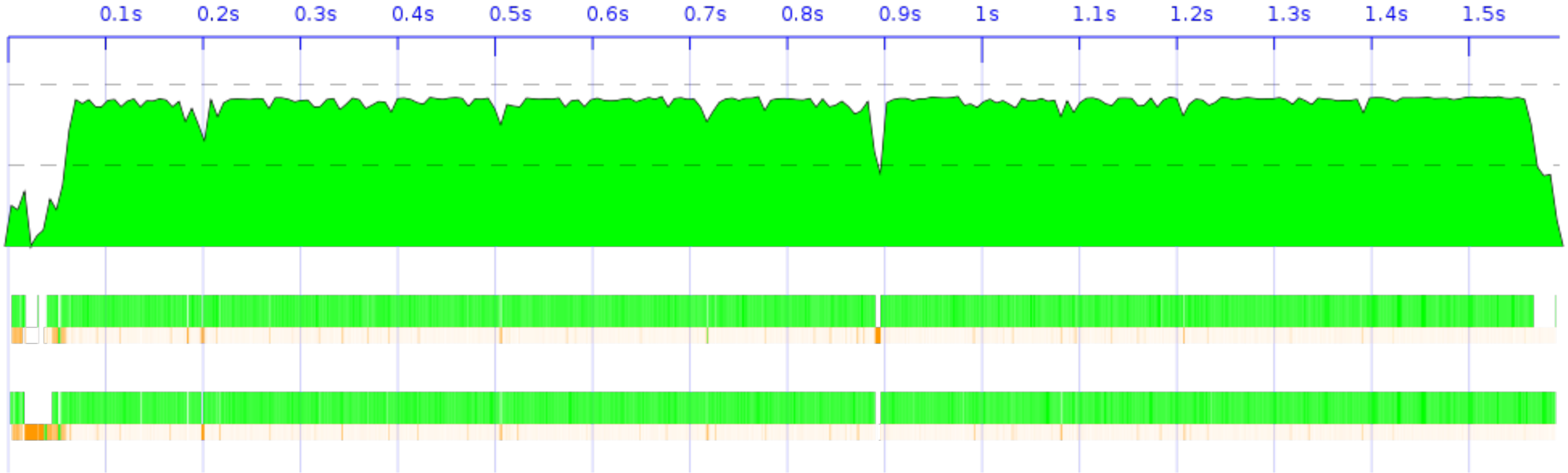
Simple idea: parallel map

```
paraMap:: (a -> b) -> [a] -> [b]
paraMap f [] = []
paraMap f (x:xs) = let y  = deep $ f x
                       ys = paraMap f xs
                   in y `par` ys `pseq` y : ys
```

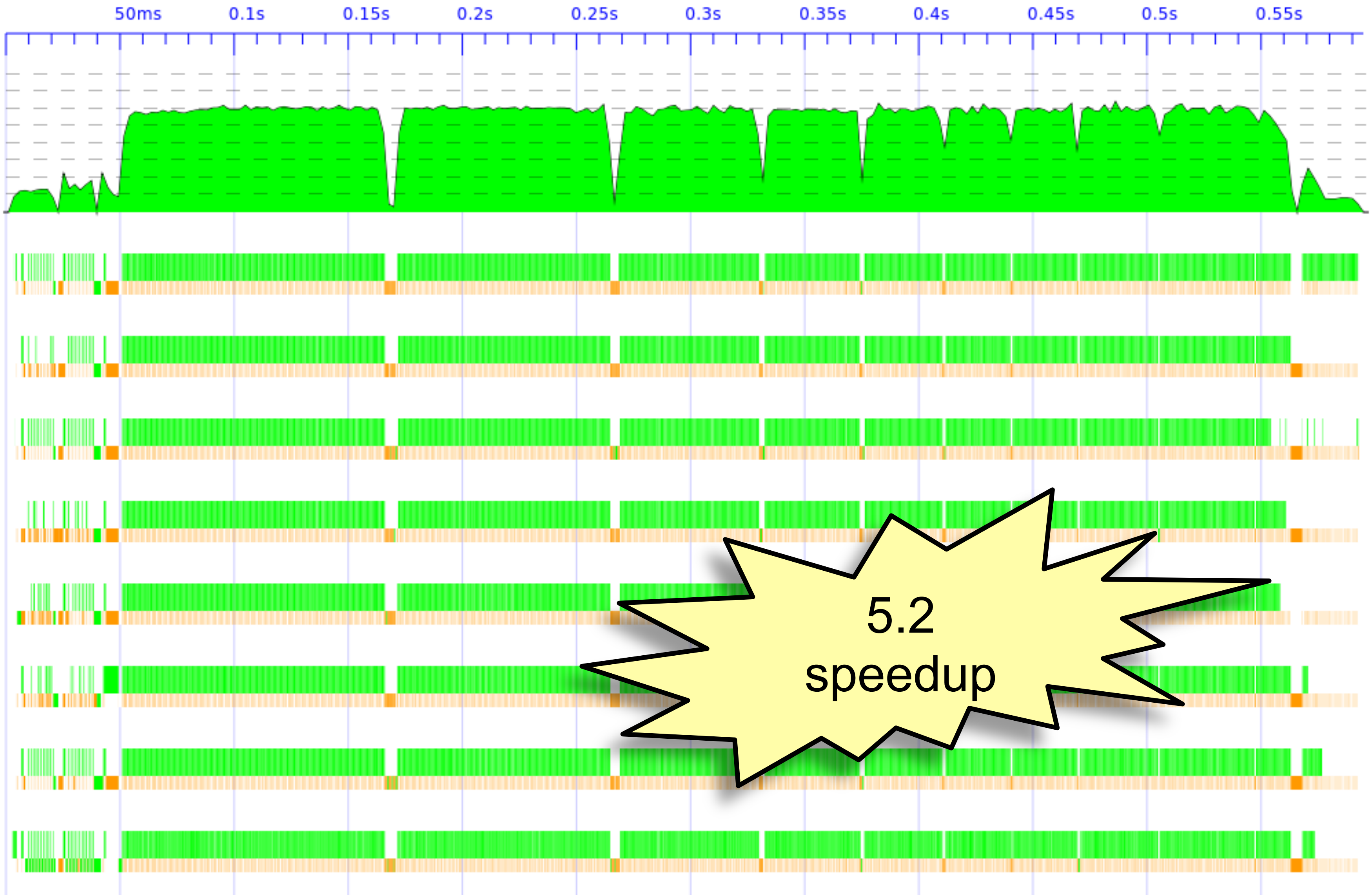
places each call to f in a new spark – to be evaluated deeply!

```
main = do
    grids <- fmap lines $ readFile f
    let solutions = paraMap solve grids
    print $ all isJust solutions
```

Result (2 cores)



Same code on an 8-core box



Evaluation Strategies

par and pseq are low level

- All about sequencing computation and creation of sparks
  - monads are good for sequencing…
- Algorithm + Strategy = Parallel program
  - *strategies* as re-usable components that can be composed together
  - Clean separation of algorithm from strategy

The Eval monad

```haskell
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)

Example: A parallel map

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

Create a spark to evaluate (f a) for each element a

Return the new list

The Strategy type

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

- A Strategy is...
  - A function that,
  - when applied to a value ‘a’,
  - evaluates ‘a’ to some degree
  - (possibly sparking evaluation of sub-components of ‘a’ in parallel),
  - and returns an equivalent ‘a’ in the Eval monad
- NB. the return value should be equivalent to the original

Some Basic Strategies

• r0	no evaluation
• rpar	create a parallel spark
• rdeepseq	deep evaluation

evalList

- parMap has the sparking behaviour built-in, start with a basic traversal in the Eval monad:

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

evalList

```
-- Earlier example could be defined as
rParaMap f =
  parList (rpar `dot` rdeepseq) . map f

parList f = evalList (rpar `dot` f)
  where s1 `dot` s2 = s1 . runEval . s2
```

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
```

- Idea: `using` strategies should always be a performance annotation.
  - need to check that x `using` s == x

Using Strategies

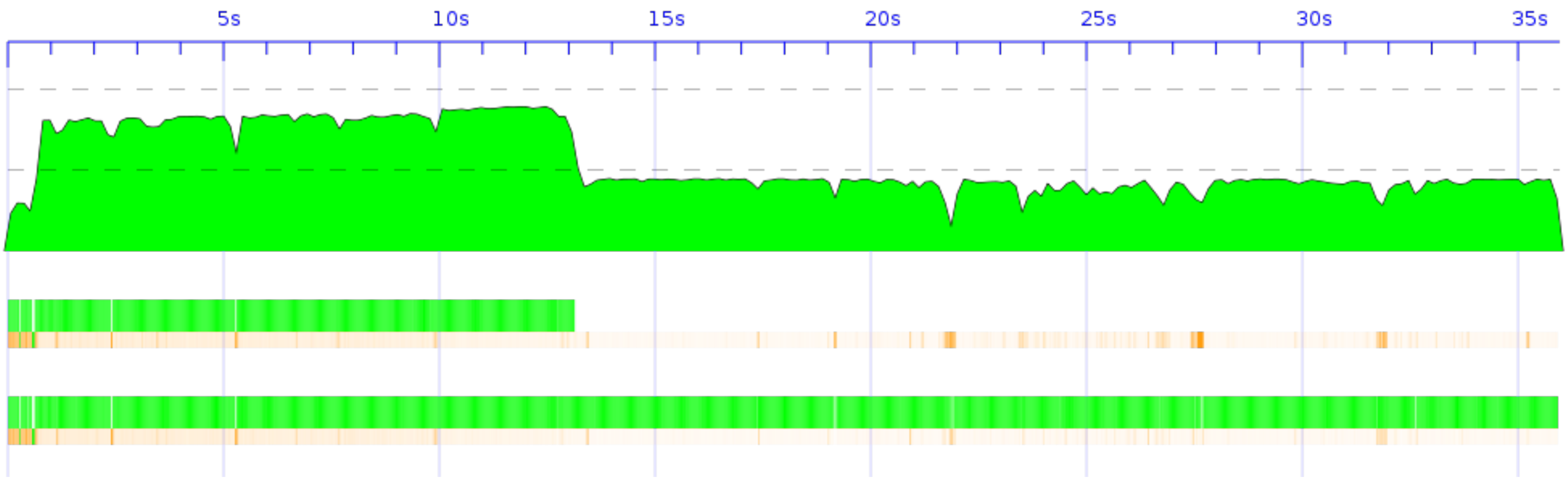
```
main = do
    grids <- fmap lines $ readFile f
    let solutions = map solve grids `using` strat
    print $ all isJust solutions
```

```
strat = evalList (rpar `dot` rdeepseq)
```

- Note: this modularity depends crucially on lazy evaluation – otherwise strat would be too late to have any control (the term would already be evaluated!)

What if the file is BIG

- 1000 -> 16000 sudokus
- Spark pool buffer exceeded – lost sparks –



chunkList Strategy

- Strategy idea – spark chunks of n elements (fewer sparks)

```
strat = chunkList 100 (rpar `dot` rdeepseq)

chunkList n strat =
   fmap concat . evalList strat . chunks
     where chunks = takeWhile (not.null)
                  . map (take n)
                  . iterate (drop n)

prop_chunkList k xs = k > 0 ==>
     xs == xs `using` chunkList k rdeepseq
```

chunkList is parListChunk

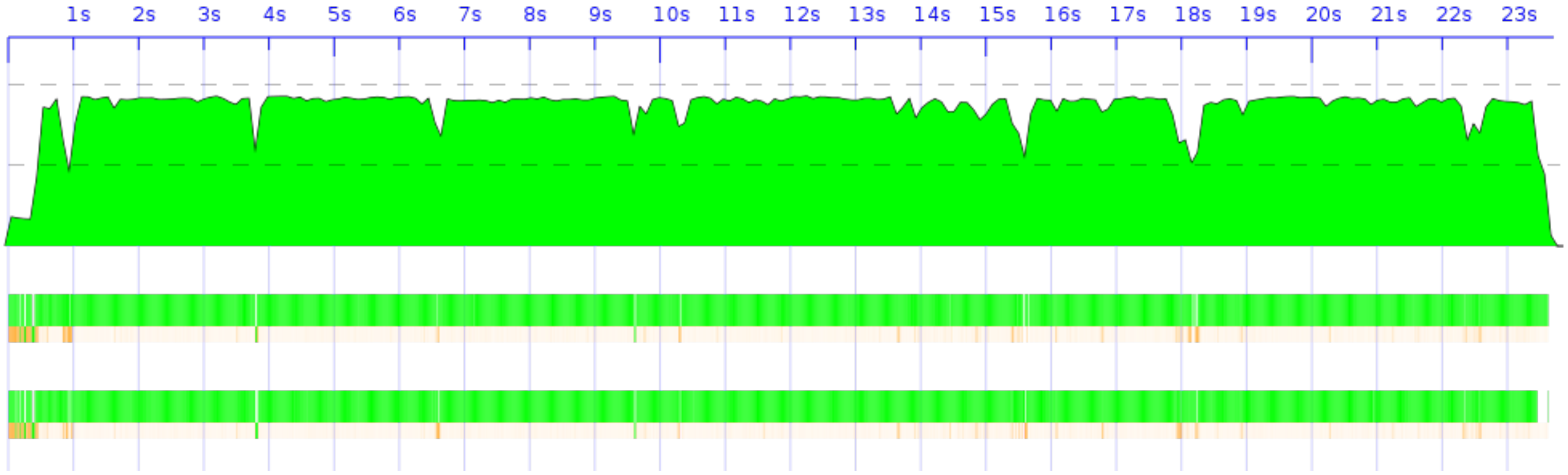
- This function already exists in the strategies library (I missed it first time!)

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

Source

Divides a list into chunks, and applies the strategy `evalList strat` to each chunk in parallel.

16000 in <24s


1s
2s
3s
4s
5s
6s
7s
8s
9s
10s
11s
12s
13s
14s
15s
16s
17s
18s
19s
20s
21s
22s
23s

Summary

- Strategies, in theory:
  - *Algorithm + Strategy = Parallelism*
- Strategies, in practice (sometimes):
  - *Algorithm + Strategy = No Parallelism*
- laziness is the magic ingredient that bestows modularity, but laziness can be tricky to deal with.

Where to look next

- Other alternatives are emerging, see e.g.
  - The Par monad: abandon modularity via laziness for more explicit concurrency
  - Data-parallel Haskell – operations on bulk data (think GPU’s – thousands of cores)

Further Reading

- Many slides here adapted from Simon Marlow's CEFP summer school slides
- http://research.microsoft.com/en-us/people/simonmar
  - /par-tutorial.pdf
  - /papers/strategies.pdf
- haskell.org/haskellwiki/ThreadScope