Parallel & Concurrent Haskell 1: Basic Pure Parallelism

Simon Marlow

(Microsoft Research, Cambridge, UK)
Haskell’s philosophy

• We want to give you the right tool for the job
  – Even if that means having many tools
  – so unlike some languages that focus on just one parallel programming model (e.g. CSP or message-passing) in Haskell there are lots to choose from
Haskell’s philosophy

• We want to give you the right tool for the job
  – Even if that means having many tools
  – so unlike some languages that focus on just one parallel programming model (e.g. CSP or message-passing) in Haskell there are lots to choose from

• But the guiding principle is
  – provide *minimal* built-in functionality
  – so that we can give a simple semantics
  – then implement nice abstraction layers on top
Haskell’s philosophy

• We want to give you the right tool for the job
  – Even if that means having many tools
  – so unlike some languages that focus on just one parallel programming model (e.g. CSP or message-passing) in Haskell there are lots to choose from

• But the guiding principle is
  – provide \textit{minimal} built-in functionality
  – so that we can give a simple semantics
  – then implement nice abstraction layers on top

• Problem: how do you, the programmer, decide which tool (API) you need?
First, we divide the landscape in two: *Parallel* and *Concurrent* applications/programming models.
First, we divide the landscape in two: *Parallel* and *Concurrent* applications/programming models

What’s the difference?
Parallelism vs. Concurrency

- Multiple cores for *performance*
- Multiple threads for *modularity of interaction*

- Parallel Haskell
- Concurrent Haskell
Parallelism vs. Concurrency

Multiple cores for performance

Multiple threads for modularity of interaction

Parallel Haskell

Concurrent Haskell
Parallelism vs. Concurrency

• Primary distinguishing feature of Parallel Haskell: determinism
  – The program always does the same thing, but may run faster when given multiple cores to run on.
  – No race conditions or deadlocks
  – add parallelism without sacrificing correctness
  – Parallelism is used to speed up pure (non-IO monad) Haskell code
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)
We have a lot of ground to cover...

1. Basic pure parallelism
2. The Par Monad
3. Concurrent Haskell
4. Software Transactional Memory
5. Server applications
6. Distributed programming
7. GPU programming
I. Parallel Haskell

- In this part of the course, you will learn how to:
  - Do basic parallelism:
    - compile and run a Haskell program, and measure its performance
    - parallelise a simple Haskell program (a Sudoku solver)
    - use ThreadScope to profile parallel execution
    - do dynamic partitioning
    - measure parallel speedup
      - use Amdahl’s law to calculate possible speedup
  - Work with 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

```
import Sudoku

solve :: String -> Maybe Grid

......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.........
```
Solving Sudoku problems

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

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

solve :: String -> Maybe Grid
Compile the program...

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

$ ./sudoku1 sudoku17.1000.txt +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)

...  


Now to parallelise it...

• Doing parallel computation entails specifying coordination in some way – compute A in parallel with B

• This is a constraint on *evaluation order*

• But by design, Haskell *does not have a specified evaluation order*

• So we need to add something to the language to express constraints on evaluation order
The Eval monad

- **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)
- We want to do a in parallel with b.
- Which of the following is the best?

\[
\begin{align*}
\text{do} & \\
a' & \leftarrow \text{rpar } a \\
b' & \leftarrow \text{rpar } b \\
\text{return } (a',b')
\end{align*}
\]
• We want to do a in parallel with b.
• Which of the following is the best?
• We want to do a in parallel with b.
• Which of the following is the best?

\[
\begin{align*}
\text{do} & \quad 
\begin{align*}
a' & \leftarrow \text{rpar } a \\
b' & \leftarrow \text{rpar } b \\
\text{return } (a',b')
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{do} & \quad 
\begin{align*}
a' & \leftarrow \text{rpar } a \\
b' & \leftarrow \text{rseq } b \\
\text{return } (a',b')
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{do} & \quad 
\begin{align*}
a' & \leftarrow \text{rpar } a \\
b' & \leftarrow \text{rpar } b \\
\text{rseq } a' \\
\text{return } (a',b')
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{do} & \quad 
\begin{align*}
a' & \leftarrow \text{rpar } a \\
b' & \leftarrow \text{rpar } b \\
\text{rseq } a' \\
\text{rseq } b' \\
\text{return } (a',b')
\end{align*}
\end{align*}
\]
• We want to do \( a \) in parallel with \( b \).
• Which of the following is the best?
What does rpar *actually* do?

- **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 – new sparks are discarded when the pool is full. Watch out!

\[ x \leftarrow \texttt{rpar e} \]
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
```

• Now we need something like

```
runEval $ do
  as' <- rpar (map solve as)
  bs' <- rpar (map solve bs)
  rseq as'
  rseq bs'
  return (as' ++ bs')
```
runEval $ do
  as’ <- rpar (map solve as)
  bs’ <- rpar (map solve bs)
  rseq as’
  rseq bs’
  return (as’ ++ bs’)

- **rpar** evaluates its argument to Weak Head Normal Form (WHNF)

- **what is 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 to go deeper

• provided by the ‘deepseq’ package
• `force` fully evaluates a nested data structure and returns it
  – e.g. a list: the list 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

```haskell
module Control.DeepSeq ( .. ) where

class NFData a where
    rnf :: a -> ()

deepseq :: NFData a => a -> b -> b
    deepseq a b = rnf a `seq` b

force :: NFData a => a -> a
    force a = deepseq a a
```

We need this
Ok, adding force

runEval $ do
  as' <- rpar (force (map solve as))
  bs' <- rpar (force (map solve bs))
  rseq as'
  rseq bs'
  return (as' ++ bs')

• Now we just need to evaluate this at the top level in ‘main’:

print $ length $ filter isJust $ runEval $ do
  as' <- rpar (force (map solve as))
  ...

Let’s try it...

```bash
$ ghc --make -O2 sudoku2.hs -rtsopts -threaded
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
$
```
Run it on one processor first

```bash
$ ./sudoku2 sudoku17.1000.txt +RTS -s
./sudoku2 sudoku17.1000.txt +RTS -s
1000
   2,400,398,952 bytes allocated in the heap
    48,900,472 bytes copied during GC
   3,280,616 bytes maximum residency (7 sample(s))
     379,624 bytes maximum slop
     11 MB total memory in use (0 MB lost due to fragmentation)

... 

INIT time  0.00s  (  0.00s elapsed)
MUT   time  2.91s  (  2.91s elapsed)
   GC    time  0.19s  (  0.19s elapsed)
EXIT  time  0.00s  (  0.00s elapsed)
Total time  3.09s  (  3.09s elapsed)
...
```

A tiny bit slower (was 3.06 before). Splitting and reconstructing the list has some overhead.
Runtime results...

$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s
 2,400,125,664 bytes allocated in the heap
  48,845,008 bytes copied during GC
  2,617,120 bytes maximum residency (7 sample(s))
   313,496 bytes maximum slop
         9 MB total memory in use (0 MB lost due to fragmentation)

Generation 0:  2975 collections,  2974 parallel,  1.04s,  0.15s elapsed
Generation 1:     7 collections,     7 parallel,  0.05s,  0.02s elapsed

Parallel GC work balance: 1.52 (6087267 / 3999565, ideal 2)

SPARKS: 2 (1 converted, 0 pruned)

INIT time  0.00s  (  0.00s elapsed)
MUT time   2.21s  (  1.80s elapsed)
GC time    1.08s  (  0.17s elapsed)
EXIT time  0.00s  (  0.00s elapsed)
Total time 3.29s  (  1.97s elapsed)
Calculating Speedup

• Calculating speedup with 2 processors:
  – Elapsed time (1 proc) / Elapsed Time (2 procs)
  – NB. not CPU time (2 procs) / Elapsed (2 procs)!
  – NB. compare against sequential program, not parallel program running on 1 proc
    • why? introducing parallelism may add some overhead compared to the sequential version

• Speedup for sudoku2: 3.06/1.97 = 1.55
  – not great...
Why not 2?

• there are two reasons for lack of parallel speedup:
  – less than 100% utilisation (some processors idle for part of the time)
  – extra overhead in the parallel version
• Each of these has many possible causes...
A menu of ways to go wrong

• less than 100% utilisation
  – parallelism was not created, or was discarded
  – algorithm not fully parallelised – residual sequential computation
  – uneven work loads

• extra overhead in the parallel version
  – overheads from rpar, work-stealing, force, ...
  – larger memory requirements leads to GC overhead

• low-level issues that are Simon’s problem:
  – poor scheduling
  – communication latency
  – GC synchronisation
  – duplicating work
  – poor locality, cache effects
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

```
grc -O2 sudoku2.hs -threaded -rtsopts -eventlog
gc /sudoku2 sudoku17.1000.txt +RTS -N2 -l1
threadscope sudoku2.eventlog
```

- -eventlog has very little effect on runtime
  – important for profiling parallelism
Uneven workloads...

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

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

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

• 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

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

• Dynamic partitioning involves
  – dividing the work into smaller units
  – assigning work units to processors dynamically at runtime using a *scheduler*
  – good for irregular problems and varying number of processors

• GHC’s runtime system provides spark pools to track the work units, and a work-stealing scheduler to assign them to processors

• So all we need to do is use smaller tasks and more sparks, and we get dynamic partitioning
Revisiting Sudoku...

• So previously we had this:

```haskell
runEval $ do
  a <- rpar (force (map solve as))
  b <- rpar (force (map solve bs))
  ...
```

• We want to push `rpar` down into the `map` – so each call to `solve` will be a separate spark
A parallel map

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

Create a spark to evaluate (f a) for each element a
Return the new list

• Provided by Control.Parallel.Strategies
• Also:  parMap f xs = mapM (rpar . f) xs
Putting it together...

- Code is simpler than the static partitioning version!

```
runEval $ parMap solve grids
```
Results

/sudoku3 sudoku17.1000.txt +RTS -s -N2 -ls
2,401,880,544 bytes allocated in the heap
49,256,128 bytes copied during GC
2,144,728 bytes maximum residency (13 sample(s))
198,944 bytes maximum slop
7 MB total memory in use (0 MB lost due to fragmentation)

Generation 0: 2495 collections, 2494 parallel, 1.21s, 0.17s elapsed
Generation 1: 13 collections, 13 parallel, 0.06s, 0.02s elapsed

Parallel GC work balance: 1.64 (6139564 / 3750823, ideal 2)

SPARKS: 1000 (1000 converted, 0 pruned)

INIT time 0.00s ( 0.00s elapsed)
MUT time 2.19s ( 1.55s elapsed)
GC time 1.27s ( 0.19s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 3.46s ( 1.74s elapsed)

Now 1.7 speedup
5.2 speedup
break...
• Lots of GC
• One core doing all the GC work
  – indicates one core generating lots of data
main :: IO ()
main = do
  [f] <- getArgs
  grids <- fmap lines $ readFile f
  print $ length $ filter isJust $
    runEval $ parMap solve grids

• Are there any sequential parts of this program?
main :: IO ()
main = do
  [f] <- getArgs
  grids <- fmap lines $ readFile f
  print $ length $ filter isJust $ runEval $ parMap solve grids

• Are there any sequential parts of this program?
• readFile and lines are not parallelised
• Suppose we force the sequential parts to happen first...

```haskell
main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    evaluate (length grids)
    print $ length $ filter isJust $ runEval $ parMap solve grids
```
The image shows a timeline analysis using the ThreadScope tool. The timeline is divided into 20ms intervals, starting from 20ms to 40ms. The activity section displays various thread events such as creating thread, running, GC, seq GC req, par GC req, migrate thread, thread wakeup, and shutdown. The events section lists specific thread events, including:

- 38106000: cap 1: running thread 3
- 38126000: cap 0: creating thread 4
- 38126000: cap 0: creating spark thread 4
- 38126000: cap 0: thread 4 is runnable
- 38200000: cap 0: running thread 4
- 38211000: cap 0: thread 4 stealing a spark from cap 1
Calculating possible speedup

• When part of the program is sequential, Amdahl’s law tells us what the maximum speedup is.

\[ \frac{1}{(1 - P) + \frac{P}{N}} \]

• P = parallel portion of runtime

• N = number of processors
Applying Amdahl’s law

- In our case:
Applying Amdahl’s law

• In our case:
  – runtime = 3.06s (NB. sequential runtime!)
Applying Amdahl’s law

• In our case:
  – runtime = 3.06s (NB. sequential runtime!)
  – non-parallel portion = 0.038s (P = 0.9876)
Applying Amdahl’s law

- In our case:
  - runtime = 3.06s (NB. sequential runtime!)
  - non-parallel portion = 0.038s (P = 0.9876)
  - N = 2, max speedup = \( \frac{1}{(1 - 0.9876) + \frac{0.9876}{2}} \)
    - \(~ 1.98\)
    - on 2 processors, maximum speedup is not affected much by this sequential portion
Applying Amdahl’s law

• In our case:
  – runtime = 3.06s (NB. sequential runtime!)
  – non-parallel portion = 0.038s (P = 0.9876)
  – N = 2, max speedup = 1 / ((1 – 0.9876) + 0.9876/2)
    • =~ 1.98
      • on 2 processors, maximum speedup is not affected much by this sequential portion
  – N = 64, max speedup = 35.93
    • on 64 processors, 38ms of sequential execution has a dramatic effect on speedup
• diminishing returns...

• See “Amdahl's Law in the Multicore Era”, Mark Hill & Michael R. Marty
• Amdahl’s law paints a bleak picture
  – speedup gets increasingly hard to achieve as we add more cores
  – returns diminish quickly when more cores are added
  – small amounts of sequential execution have a dramatic effect
  – proposed solutions include heterogeneity in the cores
    – likely to create bigger problems for programmers
• Amdahl’s law paints a bleak picture
  – speedup gets increasingly hard to achieve as we add more cores
  – returns diminish quickly when more cores are added
  – small amounts of sequential execution have a dramatic effect
  – proposed solutions include heterogeneity in the cores
    – likely to create bigger problems for programmers

• See also Gustafson’s law – the situation might not be as bleak as Amdahl’s law suggests:
  – with more processors, you can solve a bigger problem
  – the sequential portion is often fixed or grows slowly with problem size
• Amdahl’s law paints a bleak picture
  – speedup gets increasingly hard to achieve as we add more cores
  – returns diminish quickly when more cores are added
  – small amounts of sequential execution have a dramatic effect
  – proposed solutions include heterogeneity in the cores
    – likely to create bigger problems for programmers

• See also Gustafson’s law – the situation might not be as bleak as Amdahl’s law suggests:
  – with more processors, you can solve a bigger problem
  – the sequential portion is often fixed or grows slowly with problem size

• Note: in Haskell it is hard to identify the sequential parts anyway, due to lazy evaluation
Evaluation Strategies

• So far we have used Eval/rpar/rseq
  – these are quite low-level tools
  – but it’s important to understand how the underlying mechanisms work

• Now, we will raise the level of abstraction

• Goal: encapsulate parallel idioms as re-usable components that can be composed together.
The Strategy type

• **Strategy** \(a\) 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 observably equivalent to the original
  – (why not the same? we’ll come back to that...)
Example...

```haskell
parList :: Strategy [a]
```

- A **Strategy** on lists that sparks each element of the list
Example...

parList :: Strategy [a]

- A **Strategy** on lists that sparks each element of the list
- This is usually not sufficient – suppose we want to evaluate the elements fully (e.g. with **force**), or do **parList** on nested lists.
Example...

parList :: Strategy \[a\]

• A **Strategy** on lists that sparks each element of the list

• This is usually not sufficient – suppose we want to evaluate the elements fully (e.g. with **force**), or do **parList** on nested lists.

• So we parameterise **parList** over the **Strategy** to apply to the elements:
Example...

```
parList :: Strategy [a]
```

• A **Strategy** on lists that sparks each element of the list

• This is usually not sufficient – suppose we want to evaluate the elements fully (e.g. with `force`), or do `parList` on nested lists.

• So we parameterise `parList` over the **Strategy** to apply to the elements:

```
parList :: Strategy a -> Strategy [a]
```
This is what we mean by “composable”:
– given a Strategy on the list elements,
– **parList** gives us a Strategy on lists of those elements

We have some simple Strategies already:

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

rpar :: a -> Eval a  -- same as Strategy a
rseq :: a -> Eval a  -- ditto
```

here’s a couple more:

```
r0 :: Strategy a
rdeepseq :: NFData a -> Strategy a
```

so here’s a simple composition:

```
parList rdeepseq :: Strategy [a]
```
Strategies are easy to define

• We have the building blocks:

```haskell
type Strategy a = a -> Eval a
parList :: Strategy a -> Strategy [a]

rpar :: a -> Eval a
```
Strategies are easy to define

- We have the building blocks:

```haskell
type Strategy a = a -> Eval a
parList :: Strategy a -> Strategy [a]

rpar :: a -> Eval a

parList :: (a -> Eval a) -> [a] -> Eval [a]
    -- same as Strategy a -> Strategy [a]

parList s []     = return []
parList s (x:xs) = do
    x'     <- rpar (runEval (s x))
    xs'    <- parList s xs
    return (x':xs')
```
Let’s generalise...

• **parList** has **rpar** built-in, but we might not want that. Let’s make a version without the **rpar**:

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

Now we can define **parList** in terms of **trainList**:

```haskell
parList f = trainList (rparWith f)
```

```haskell
rparWith :: Strategy a -> Strategy a
rparWith s a = rpar (runEval (s a))
```
Let’s make a general Strategy on pairs:

```
let evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
  evalTuple2 sa sb (a,b) = do
    a' <- sa a
    b' <- sb b
    return (a',b')
```

Example: a Strategy on a pair that evaluates the first component and sparks the second:

```
evalTuple2 rseq rpar :: Strategy (a,b)
```

(left-then-right ordering is built into evalTuple2, if you want the other ordering you have to define a different evalTuple2)
So far:

```haskell
rpar, rseq, r0 :: Strategy a
rdeepseq :: NFData a => Strategy a

rparwith :: Strategy a -> Strategy a

evalList :: Strategy a -> Strategy [a]
evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
```

Here are some example Strategies. What do they do?

```haskell
evalList (evalTuple2 rpar r0) :: Strategy [(a,b)]

evalList (rparwith (evalTuple2 rseq rseq)) :: Strategy [(a,b)]

evalList (evalTuple2 rdeepseq rpar) :: Strategy [(a,b)]

evalList (evalList rpar) :: Strategy [[a]]
evalList (rparwith (evalList rseq)) :: Strategy [[a]]
evalList (rparwith (evalList rpar)) :: Strategy [[a]]
```
How do we *use* a Strategy?

type `Strategy a = a -> Eval a`
How do we *use* a Strategy?

- We could just use `runEval`
- But this is better:

```haskell
x `using` s = runEval (s x)
```
How do we use a Strategy?

- We could just use `runEval`
- But this is better:
  ```haskell
  type Strategy a = a -> Eval a
  
  x `using` s = runEval (s x)
  
  e.g.
  myList `using` parList rdeepseq
  ```
How do we use a Strategy?

- We could just use `runEval`
- But this is better:
  ```haskell
  x `using` s = runEval (s x)
  ```
- e.g.
  ```haskell
  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?

1. It relies on a **Strategy** 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 `_|_`)
But we wanted **parMap**

- Earlier we used **parMap** to parallelise Sudoku
- But **parMap** is a combination of two concepts:
  - The *algorithm*, ‘map’
  - The *parallelism*, ‘parList’

\[
\text{parMap } f \ x = \text{map } f \ xs \ `\text{using}\` \text{parList } \text{rseq}
\]

- With Strategies, the algorithm can be separated from the parallelism.
  - The algorithm produces a (lazy) result
  - A **Strategy** filters the result, but does not do any computation – it returns the same result.
• So a nicer way to write the Sudoku example is:

```haskell
main :: IO ()
main = do
  [f] <- getArgs
  grids <- fmap lines $ readFile f
  evaluate (length grids)
  evaluate $ force $ 
    map solve grids `using` parList rseq
```

• Here the parallelism is modular
Recap...

- Eval monad, for expressing evaluation order and sparking:

```haskell
data Eval a -- instance Monad Eval

runEval :: Eval a -> a

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

- Strategies,
  - built using Eval, rpar, rseq
  - express compositional parallelism over data structures
  - modular: parallelism separate from algorithm

```haskell
type Strategy a = a -> Eval a
using :: a -> Strategy a -> a
```
extra stuff
Generalise further...

• In fact, evalList already exists for arbitrary data types in the form of ‘traverse’.

\[
evalTraversable :: \text{Traversable } t \Rightarrow \text{Strategy } a \rightarrow \text{Strategy } (t \ a)
\]

\[
evalTraversable = traverse
\]

\[
evalList = evalTraversable
\]

• So, building Strategies for arbitrary data structures is easy, given an instance of \textbf{Traversable}.

• (not necessary to understand \textbf{Traversable} here, just be aware that many Strategies are just generic traversals in the \textbf{Eval} monad).
By why *do* Strategies return a value?

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

- Spark pool points to *(runEval (s x))*
- If nothing else points to this expression, the runtime will discard the spark, on the grounds that it is not required
- *Always keep hold of the return value of rpar*
- (see the notes for more details on this)