# (Deterministic) Parallelism in Haskell

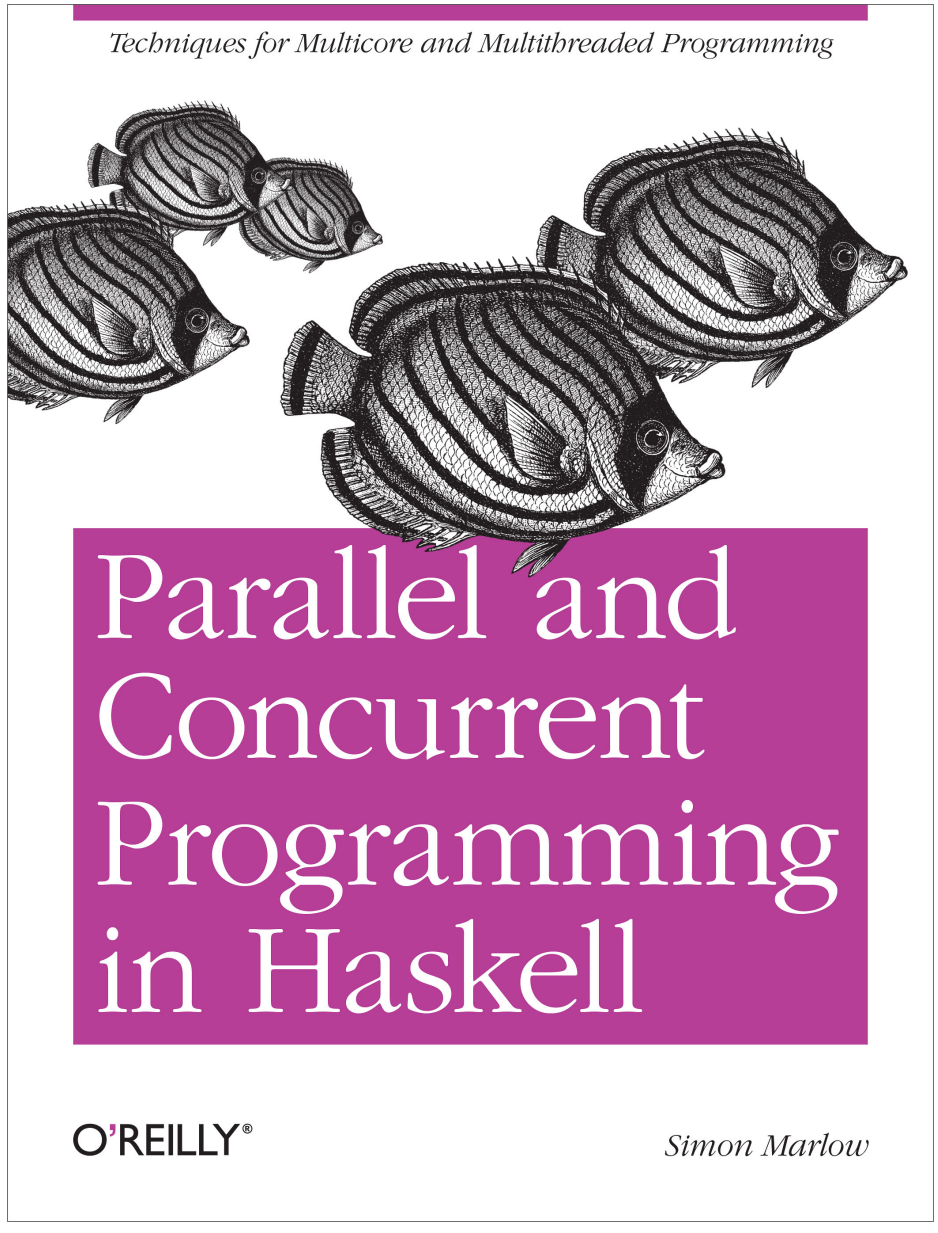
Marek Imiełowski

22.04.2015

# Section 1

# Introduction

# Main reference



# Basic definitions

## Parallelism

Running (parts of) programs at the same time on multiple cores (or nodes) in order to perform computations more quickly.

## Concurrency

A technique of structuring a program as if it has many independent threads of control.

# Automatic parallelism?

Still a future goal.

# Deterministic parallelism

We call a program **deterministic** if its result is independent of the number of cores it is being run on and the individual run of the program.

Deterministic parallelism is quite unique to Haskell (or, more broadly, purely functional programming), but it extremely simpifies writing and reasoning about parallel programs.

# The landscape of parallel Haskell

Deterministic approaches:

- semi-explicit parallelism with evaluation strategies (`parallel`)
- dataflow parallelism (`monad-par`)
- nested data parallelism (`DPH`)
- flat data parallelism with regular arrays (`repa`)
- embedded language for flat data parallelism (`accelerate`)

Non-deterministic approaches:

- concurrency primitives (`forkIO`,...)
- dataflow parallelism with side effects (`ParIO`)
- asynchronous computations (`async`)
- Cloud Haskell (`distributed-process`)

# Today

1. Basic parallelism with `Eval` monad and a bit about toolset.
2. Evaluation strategies for `Eval`.
3. Dataflow parallelism with `Par` monad.

# Section 2

# Basic parallelism

# The `Eval` monad, `rpar`, and `rseq`

The module `Control.Parallel.Strategies` provides the following inteface:

```
data Eval a
instance Monad Eval

runEval ::  Eval a -> a

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

Let's see this in practice!

# Compilation and running

Compile with:

```
$ ghc -O2 -threaded -rtsopts -eventlog rpar.hs
```

Run with:

```
$ ./rpar 1 +RTS -N2 -s -l
```

# Flags explanation

Compiler flags:

- `-O2` enables optimisation
- `-threaded` links in the threaded run-time system
- `-rtsopts` allows configuration of run-time system at run-time
- `-eventlog` allows eventlog generation for debugging and threadscope

Run-time system flags:

- `-Nx` runs on x cores (`-N` runs on all available ones)
- `-s` produces run-time statistics
- `-l` generates an eventlog for debugging

# Section 3

# Evaluation Strategies

# Basic Strategies

Evaluation Strategies is an abstraction built on top of the `Eval` monad, that allows larger parallel specifications to be built in a compositional way.

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

Some simple strategies:

```
rseq ::  Strategy a
rpar ::  Strategy a

r0 ::  Strategy a
r0 x = return x

rdeepseq ::  NFData a = > Strategy a
rdeepseq x = rseq (deep x)
```

# Basic Strategies, cont.

Ideally we would like to separate algorithm and description of parallelism:

```
using ::  a -> Strategy a -> a
x `using` s = runEval (s x)
```

See article: P. W. Trinder et al., *Algorithm + Strategy = Parallelism*.

# A Strategy for evaluating list in parallel

A function parMap that would map a function over a list in parallel can be expressed as this type of composition:

```
parMap f xs = map f xs `using` parList rseq
```

Where a Strategy on lists is defiened as follows:

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

# Example: K-Means problem

The goal is to partition a set of data points into clusters.

The most well-known heuristic finds a solution by iteratively improving an initial guess, as follows:

1. Pick an initial set of clusters by randomly assigning each point in the data set to a cluster.
2. Find the centroid of each cluster (the average of all the points in the cluster).
3. Assign each point to the cluster to which it is closest, this gives a new set of clusters.
4. Repeat steps 2–3 until the set of clusters stabilises.

# Haskell code for K-Means

# Parallelising K-Means

# Section 4

# Dataflow Parallelism with Par monad

# Dataflow Parallelism with Par monad

The interface of `monad-par` library is based around a following monad:

```
newtype Par a
instance Applicative Par
instance Monad Par

runPar ::  Par a -> a
```

We can create parallel tasks with `fork`:

```
fork ::  Par () -> Par ()
```

# IVar type

Values can be passed between `Par` computations using the `IVar` type:

```
data IVar a -- instance Eq


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

See article: S. Marlow, *A monad for deterministic parallelism*.

# parMap with Par monad

First we build a simple abstraction for a parallel computation that returns a result:

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

# parMap with Par monad

Parallel map consists of calling spawn to apply the function to each element of the list and then waiting for the results:

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

# Parallelising K-Means

(now with Par monad)

Questions?