Parallel Programming in Haskell

- Parallelism has long been discussed in functional programming fields due to the immutable data and side-effect free computation
- No worry of deadlocks, starvations, synchronization, and locks
- Have to pay attention to execution granularity though

Fibonacci Sequence calculator

A small example

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

main = print $ fib 37
```

Spark Parallelism

- Control.Parallel library
- This gives us the par operation, allowing us to signal sites of potential parallelism

    ```haskell
    par :: a -> b -> b
    ```

    Semantically par x y is equivalent to just y, but the runtime is allowed to use it as a hint.

    Next try

    ```haskell
    import Control.Parallel

    fib :: Integer -> Integer
    fib n | n < 2 = 1
    fib n = par nf ( fib (n-1) + nf )
            where nf = fib (n-2)

    main = print $ fib 37
    ```
-
- **This makes the program slower.**
- Why?
    - Par nf (fib (n-1) + nf)
    - In parallel evaluation, the RHS depends on the nf value, when the nf value is computed, a new nf dependency is introduced in the evaluation of fib (n-1)
    - **This is caused by the order in which (+) evaluates its arguments**

We shouldn't need to consider something as low level as the order in which (+) handles its arguments…

- pseq
- pseq :: a -> b -> b
- Evaluate a before b, returning b

    ```haskell
    fib n | n < 2 = 1
    fib n = par nf1 (pseq nf2 (nf1 + nf2))
            where nf1 = fib (n-1)
                  nf2 = fib (n-2)
    ```
-
    - Evaluate nf1 while at the same time (evaluate nf2 and then return nf1 + nf2)

- **Much faster**

Further Tweaking

- Overheads can dominate after a while
- Limit the number of new threads to allow more even distribution of work

```haskell
import Control.Parallel

-- Sequential version of fib, when we want to avoid parallelism
sfib :: Integer -> Integer
sfib n | n < 2 = 1
sfib n = sfib (n-1) + sfib (n-2)

fib :: Integer -> Integer -> Integer
fib 0 n = sfib n
fib _ n | n < 2 = 1
fib d n = par nf1 (pseq nf2 (nf1 + nf2))
          where nf1 = fib (d-1) (n-1)
                nf2 = fib (d-1) (n-2)

main = print $ fib 3 37
```

- **Parameterised thread count**
    - **When thread count hits zero, evaluate remaining section of the numbers sequentially**

The Eval Monad

- Control.Parallel has more to it
- **Can separate algorithm and evaluation strategy**
    - **Eval Monad**
    - Semantically an Identity monad, in the same way that par is an identity function

```haskell
runEval :: Eval a -> a

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

- rpar : My argument could be evaluated in parallel
- rseq : Evaluate my argument and wait for the result

```haskell
fib d n = runEval $ do
   nf1 <- rpar $ fib (d-1) (n-1)
   nf2 <- rseq $ fib (d-1) (n-2)
   return $ nf1 + nf2
```

- The **Control.Parallel.Strategies** module defines many utilities for easily exploiting parallelism
    - **Strategies for specifying strictness (play the role of seq)**
    - Higher level functions for applying these (**parList, parListChunk, etc)** which can model algorithms like map-reduce
- **Example of Strategy -** Parallelizing a Sudoku Solver
    - (Assuming the existence of a solver: solve :: String -> Maybe Grid)

```haskell
import Sudoku
import Control.Exception
import System.Environment
import Data.Maybe

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

Version 1 is sequential:

```
$ ./sudoku-par1  sudoku17.1000.txt +RTS -N4 -s

...
  Parallel GC work balance: 1.87% (serial 0%, perfect 100%)
...
```

Now let's add some basic parallelism

```haskell
main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f

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

        solutions = runEval $ do
                      as' <- rpar (force (map solve as))
                      bs' <- rpar ( force ( map solve bs))
                      _ <- rseq as'
                      _ <- rseq bs'
                      return ( as' ++ bs' )
    print (length (filter isJust solutions))
```

    -
    - **Version 2 is far more efficient**
- **We can build Utilities in the Eval Monad**

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

```haskell
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f

  let puzzles   = lines file
      solutions = runEval (parallelMap solve puzzles)

  print (length (filter isJust solutions))
```

    -
- The library provides lots of pre-defined utilities:
    - Strategies: Control.Parallel.Strategies

    - Evaluation strategies represent a parameterized hof to capture attempts to introduce parallelism
    - **Type Strategy a = a -> Eval a**
- **Evaluate pairs in parallel**

```haskell
parPair :: Strategy (a,b)

parPair (a,b) = do a' <- rpar a
                   b' <- rpar b
                   return (a',b')
```

    -
- **Nice way to apply these strategies: `using`**

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

    -
- **Strategies in the library are parameterised to allow us more control**
    -

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

which allows us to make various pairwise evaluation strategies:

```haskell
parPair = evalPair rpar rpar
parSeqPair = evalPair rpar rseq
```

    -
    -
    -

Available library strategies

    - `r0 :: Strategy a`
    - `rseq :: Strategy a`
    - `rpar :: Strategy a`
    - `rparWith :: Strategy a -> Strategy a`

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

the `NFData` class represents data which can be fully evaluated.

```
rnf :: a -> ()
rnf a = a `seq` ()
```

- `rparWith :: Strategy a -> Strategy a`

This can "wrap" an `rpar` around it's argument strategy. For example:

```
parPair sa sb = evalPair (rparWith sa) (rparWith sb)
```

Instead of the very parallel `parallelMap` we developed, the library has a parameterised implementation using some utilities:

```
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 (rparWith s)

parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap s f = (map f) `using` (parList s) . map f
```

Our Sudoku solver could have been written

```
let solutions = map solve puzzles `using` parList rseq
```

Many more utilities and strategies in the `Control.Parallel` module