Parallelism in Haskell

ECS713

Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary
University of London

Techniques for Multicore and Multithreaded Programming

Parallel and Concurrent Programming in Haskell

O'REILLY®

Simon Marlow

Parallel vs Concurrent

Parallel	Concurrent
Computing a value faster	Combining interacting systems
Task broken into smaller (similar) sub-tasks that can be processed independently	Various processes performing different tasks and communicating between them
Lives in the pure world	Lives in the IO monad
Results often combined in main program	Usually the system is not meant to terminate

fib.hs

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = do
    let n = fib 43
    print n
```

E.g. **`fib 10`** relies on **`fib 9`** and **`fib 8`**


```
lecture 11» stack ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.3
lecture 11» stack ghc fib.hs
[1 of 1] Compiling Main ( fib.hs, fib.o )
Linking fib ...
lecture 11» time ./fib
701408733
./fib  27.80s user 0.13s system 98% cpu 28.248 total
```

Control.Parallel

Parallel Constructs

par :: a -> b -> b | infixr 0 | # Source

Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

`a` `` `par` `` `b` is exactly equivalent semantically to `b`.

`par` is generally used when the value of `a` is likely to be required later, but not immediately. Also it is a good idea to ensure that `a` is not a trivial computation, otherwise the cost of spawning it in parallel overshadows the benefits obtained by running it in parallel.

normally the value of the first argument is needed in the second argument

Control.Parallel

Parallel Constructs

pseq :: a -> b -> b | infixr 0 | # Source

Semantically identical to `seq`, but with a subtle operational difference: `seq` is strict in both its arguments, so the compiler may, for example, rearrange `a` `` `seq` `` `b` into `b` `` `seq` `` `a` `` `seq` `` `b`. This is normally no problem when using `seq` to express strictness, but it can be a problem when annotating code for parallelism, because we need more control over the order of evaluation; we may want to evaluate `a` before `b`, because we know that `b` has already been sparked in parallel with `par`.

fib-par.hs

```haskell
import Control.Parallel

fib :: Int -> Int -> Int
fib _ 0 = 1
fib _ 1 = 1
fib l n | l == 0 = fib l (n-1) + fib l (n-2)
fib l n | l > 0 = x `par` (y `pseq` x + y)
   where
      x = fib (l-1) (n-1)
      y = fib (l-1) (n-2)

main = print $ fib 3 43
```

Evaluate these two expressions in parallel

using -threaded

```
lecture 11» stack ghc fib-par.hs -- -threaded -rtsopts
[1 of 1] Compiling Main ( fib-par.hs, fib-par.o )
Linking fib ...
lecture 11» time ./fib-par +RTS -N4
701408733
./fib-par +RTS -N4  35.37s user 0.38s system 299% cpu 11.950 total
```

Compiling and Running


```
$ ghc -O2 -threaded -rtsopts -main-is parallel.hs
[1 of 2] Compiling Spin             ( Spin.hs, S
[2 of 2] Compiling Main             ( spinMain.h                )
Linking spinMain ...
$ ./parallel +RTS -N4 -sstderr -RTS 0 42
  We have case 0 and size 42
    496 bytes allocated in the heap
     36 b                    ng CC
```

run optimizations

use threaded version of runtime system

enable RTS options

program arguments

runtime system arguments

number of cores

show diagnostics

fib-map.hs

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = do
  let xs = [20..40]
  print $ map fib xs
```



```
lecture 11» stack ghc fib-map.hs
[1 of 1] Compiling Main ( fib-map.hs, fib-map.o )
Linking fib ...
lecture 11» time ./fib-map
[10946,17711,28657,...]
./fib-map  16.79s user 0.06s system 98% cpu 17.067 total
```

Control.Parallel.Strategies

Parallel Evaluation Strategies, or Strategies for short, provide ways to express parallel computations. Strategies have the following key features:


```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]   # Source
```


A combination of `parList` and `map`, encapsulating a common pattern:


```
parMap strat f = withStrategy (parList strat) . map f
```

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

Strategy is a monad!

`rdeepseq` fully evaluates its argument.

```
rdeepseq == evalSeq Control.Seq.rdeepseq
```

fib-map-par.hs

```haskell
import Control.Parallel.Strategies

fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = do
    let xs = [25..37]
    print $ (parMap rdeepseq) fib xs
```

using **parMap** instead of **map**


```
lecture 11» stack ghc fib-map-par.hs -- -threaded -rtsopts
[1 of 1] Compiling Main ( fib-map-par.hs, fib-map-par.o )
Linking fib ...
lecture 11» time ./fib-map-par +RTS -N4
[10946,17711,28657,...]
./fib-map-par +RTS -N4  4.59s user 0.14s system 202% cpu 2.338 total
```