EDAF95/EDAN40: Functional Programming
On Concurrency and Parallelism

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

April 24th, 2024

Topic of today: concurrency and parallelisation

References:

- O'Sullivan, Goerzen, Stewart: Real World Haskell, Chapter 24,
- Peyton Jones, Singh: A tutorial on parallel and concurrent programming in Haskell
- Chakravarty, Leshchinskiy, Peyton Jones, Keller, Marlow: Data Parallel Haskell: a status report
- Marlow, Parallel and Concurrent Programming in Haskell

Short note: Profiling

- Compile your program with the `-prof` and `-fprof-auto` options
- Run it with `+RTS` option `-p`

Example:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Short note: Profiling

- Compile your program with the `-prof` and `-fprof-auto` options
- Run it with `+RTS` option `-p`

Example:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compile and run:

```
jacek$ ghc -prof -fprof-auto -rtsopts FibProfile.hs
[1 of 1] Compiling Main     ( FibProfile.hs, FibProfile.o )
Linking FibProfile ...
jacek$ ./FibProfile +RTS -p
1346269
jacek$ less FibProfile.prof
```

Profiling, cont.

```
Sat May  4 00:13 2019 Time and Allocation Profiling Report  (Final)

   FibProfile +RTS -p -RTS

total time  =        0.33 secs   (332 ticks @ 1000 us, 1 processor)
total alloc = 409,314,936 bytes  (excludes profiling overheads)

COST CENTRE MODULE    SRC                  %time %alloc

fib         Main      FibProfile.hs:2:1-50  99.7  100.0


                                                                        individual      inherited
COST CENTRE  MODULE                SRC                  no.      entries  %time %alloc   %time %alloc

MAIN         MAIN                  <built-in>           115            0    0.0    0.0   100.0  100.0
 CAF         Main                  <entire-module>      229            0    0.0    0.0    99.7  100.0
  main       Main                  FibProfile.hs:1:1-21 230            1    0.0    0.0    99.7  100.0
   fib       Main                  FibProfile.hs:2:1-50 232      2692537   99.7  100.0    99.7  100.0
 CAF         GHC.Conc.Signal       <entire-module>      209            0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding       <entire-module>      191            0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding.Iconv <entire-module>      189            0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.FD      <entire-module>      180            0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.Text    <entire-module>      178            0    0.0    0.0     0.0    0.0
 main        Main                  FibProfile.hs:1:1-21 231            0    0.3    0.0     0.3    0.0
```

Profiling, cont.

CAF (Constant Applicative Form) = top-level thunk

Profiling, cont.

CAF (Constant Applicative Form) = top-level thunk

2nd example:

```
main = print (f 30 + g 30)
  where
    f n = fib n
    g n = fib (n `div` 2)

fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Profiling, cont.

```
Sat May  4 00:16 2019 Time and Allocation Profiling Report  (Final)

   FibProfile2 +RTS -p -RTS

total time  =        0.34 secs   (342 ticks @ 1000 us, 1 processor)
total alloc = 495,839,264 bytes  (excludes profiling overheads)

COST CENTRE MODULE    SRC                    %time %alloc

fib         Main      FibProfile2.hs:6:1-50  100.0  100.0

                                                                              individual      inherite
COST CENTRE  MODULE                SRC                        no.     entries  %time %alloc   %time %allo

MAIN         MAIN                  <built-in>                 115           0    0.0    0.0   100.0  100.
 CAF         Main                  <entire-module>            229           0    0.0    0.0   100.0  100.
  main       Main                  FibProfile2.hs:(1,1)-(4,25) 230          1    0.0    0.0   100.0  100.
   main.f    Main                  FibProfile2.hs:3:5-15      234           1    0.0    0.0    99.7   99.
    fib      Main                  FibProfile2.hs:6:1-50      235     2692537   99.7   99.9    99.7   99.
   main.g    Main                  FibProfile2.hs:4:5-25      232           1    0.0    0.0     0.3    0.
    fib      Main                  FibProfile2.hs:6:1-50      233        1973    0.3    0.1     0.3    0.
 CAF         GHC.Conc.Signal       <entire-module>            209           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Encoding       <entire-module>            191           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Encoding.Iconv <entire-module>            189           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Handle.FD      <entire-module>            180           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Handle.Text    <entire-module>            178           0    0.0    0.0     0.0    0.
 main        Main                  FibProfile2.hs:(1,1)-(4,25) 231          0    0.0    0.0     0.0    0.
```

Profiling, cont.

```
Mon May  6 10:31 2019 Time and Allocation Profiling Report  (Final)

   FibProfile2 +RTS -ls -N4 -p -RTS

total time  =        0.07 secs   (273 ticks @ 1000 us, 4 processors)
total alloc = 495,917,456 bytes  (excludes profiling overheads)

COST CENTRE MODULE    SRC                    %time %alloc
fib         Main      FibProfile2.hs:6:1-50  100.0  100.0
                                                                                   individual      inherite
COST CENTRE  MODULE                SRC                         no.     entries  %time %alloc   %time %allo

MAIN         MAIN                  <built-in>                  115           0    0.0    0.0   100.0  100.
 CAF         Main                  <entire-module>             229           0    0.0    0.0   100.0  100.
  main       Main                  FibProfile2.hs:(1,1)-(4,25) 230           1    0.0    0.0   100.0  100.
   main.f    Main                  FibProfile2.hs:3:5-15       234           1    0.0    0.0    99.6   99.
    fib      Main                  FibProfile2.hs:6:1-50       235     2692537   99.6   99.9    99.6   99.
   main.g    Main                  FibProfile2.hs:4:5-25       232           1    0.0    0.0     0.4    0.
    fib      Main                  FibProfile2.hs:6:1-50       233        1973    0.4    0.1     0.4    0.
 CAF         GHC.Conc.Signal       <entire-module>             209           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Encoding       <entire-module>             191           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Encoding.Iconv <entire-module>             189           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Handle.FD      <entire-module>             180           0    0.0    0.0     0.0    0.
 CAF         GHC.IO.Handle.Text    <entire-module>             178           0    0.0    0.0     0.0    0.
 CAF         GHC.Event.Poll        <entire-module>             132           0    0.0    0.0     0.0    0.
 CAF         GHC.Event.Thread      <entire-module>             131           0    0.0    0.0     0.0    0.
 main        Main                  FibProfile2.hs:(1,1)-(4,25) 231           0    0.0    0.0     0.0    0.
```

Profiling, cont.

A tool for visualisation: `ThreadScope`

0s 0.5s 1s 1.5s

Activity

HEC 0

HEC 1

this core stopped working

HEC - Haskell Execution Context

Profiling, cont.

- Compile your program with the `-threaded` and `-eventlog` options
- Run it with `+RTS` option `-ls`

Compile and run:

```
jacek$ ghc -rtsopts -threaded -eventlog FibProfile.hs
[1 of 1] Compiling Main     ( FibProfile.hs, FibProfile.o )
Linking FibProfile ...
jacek$ ./FibProfile +RTS -ls -N2
1346269
jacek$ threadscope FibProfile.eventlog
```

Threads

In `Control.Concurrent` module:

```
:t forkIO
forkIO :: IO () -> IO ThreadID
:m +System.Directory
forkIO (writeFile "foobar" "this is sth") >>
        doesFileExist "foobar"
```

(run `ghci` with `+RTS -N2` on my Mac)

```
import qualified Data.ByteString.Lazy as L
do
  content <- L.readFile name
  forkIO (compressFile name content)
  return ()
```

Threads, cont.

Communication between threads using synchronising variables:

```
-- ch24/MVarExample.hs
import Control.Concurrent

communicate = do
  m <- newEmptyMVar
  forkIO $ do
    v <- takeMVar m
    putStrLn ("received " ++ show v)
  putStrLn "sending"
  putMVar m "wake up!"
```

Threads, cores, etc.

1. I will NOT talk about concurrency (synchronisation, etc.) any more
2. Concurrent computations need to be requested *explicitely* in case you want to specify # of cores/processors, otherwise the default!
   - when compiling (`-threaded`), actually while linking
   - when invoking (`+RTS -Nx -RTS`)
   - possibly also in the code itself

   Weighing needs to be done in an informed way (Haskell lightweight threads `forkIO` vs. OS threads `forkOS`)
3. When the main thread finishes, all other get killed. Watch out!

Reminder: Normal forms

- NF (RNF) – normal form (reduced normal form)
- HNF – head normal form
- WHNF – weak head normal form

Describe the amount of evaluation performed:

- NF – evaluated
- WHNF – evaluated only up to the outermost constructor

Check `http://stackoverflow.com/questions/6872898/haskell-what-is-weak-head-normal-form`

Reminder: Space Leaks

Some definitions:

- *Strict* vs. *lazy* evaluation;
- Space leaks (foldl vs. foldl')

```
foldl (+) 0 (1:2:3:[])
          == foldl (+) (0 + 1)             (2:3:[])
          == foldl (+) ((0 + 1) + 2)       (3:[])
          == foldl (+) (((0 + 1) + 2) + 3) []
          ==           (((0 + 1) + 2) + 3)
```

Thunks stored until needed.

Reminder: Space Leaks

Space leaks (foldl vs. foldl’), `seq`

```
foldl’ _    zero []      = zero
foldl’ step zero (x:xs) =
    let new = step zero x
    in  new ‘seq‘ foldl’ step new xs
```

```
ghci> :type seq
seq :: a -> b -> b
```

`seq` solely forces evaluation of its first argument, returning the second one. But only up to WHNF!

Usage of seq

Problematic:

```
hiddenInside x y = someFunc (x `seq` y)
hiddenByLet x y z = let a = x `seq` someFunc y
                    in anotherFunc a z
badExpression step zero (x:xs) =
    seq (step zero x)
        (badExpression step (step zero x) xs)
```

Below are reasonable cases:

```
onTheOutside x y = x `seq` someFunc y
chained x y z = x `seq` y `seq` someFunc z
strictPair (a,b) = a `seq` b `seq` (a,b)
```

Sorting examples

An ordinary quicksort:

```
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
    where lesser  = sort [y | y <- xs, y <  x]
          greater = sort [y | y <- xs, y >= x]
sort _ = []
```

Sorting examples

A parallel version of quicksort:

```
import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x:xs) = force grtr `par` (force lesser `pseq`
                                      (lesser ++ x:grtr))
  where lesser = parSort [y | y <- xs, y <  x]
        grtr   = parSort [y | y <- xs, y >= x]
parSort _      = []
```

New stuff: `par`, `pseq` (from Control.Parallel) and `force` (later)

Sorting examples

- `par`: similar to `seq`, but allowing the first argument to be evaluated to WHNF *in parallel* with returning the second argument;
- `pseq`: evaluates its first argument *before* returning the second one;
  - `seq`: the possibility of evaluating the left arg first (unless the compiler thinks otherwise);
  - `pseq`: guarantees evaluating the left arg first.

`par` does not create a *thread* but a *spark*. The concept of *lazy future*.

Sparks

Another example to illustrate sparks:

```
mkList :: Int -> [Int]
mkList n = [1..n-1]

relprime :: Int -> Int -> Bool
relprime x y = gcd x y == 1

euler :: Int -> Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int -> Int
sumEuler = sum . (map euler) . mkList
```

Sparks, cntd.

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

And their sum

```
sumFibEuler :: Int -> Int -> Int
sumFibEuler a b = fib a + sumEuler b
```

`sumFibEuler 38 5300` seems reasonable for tests.

Parallel solutions

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (f + e)
    where
    f = fib a
    e = sumEuler b
```

Spark, but no thread. Why?

Parallel solutions 2

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (e + f)
    where
    f = fib a
    e = sumEuler b
```

Better, but pure luck! Compiler can change a + b into b + a at will!

Parallel solutions 3

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (e `pseq`(e + f))
    where
    f = fib a
    e = sumEuler b
```

(or `(e `pseq`(f + e))` as well)

Sorting examples

Knowing what to evaluate in parallel (possible interference with strict evaluation).

```
sillySort (x:xs) = grtr `par` (lesser `pseq`
                                (lesser ++ x:grtr))
    where lesser = sillySort [y | y <- xs, y <  x]
          grtr   = sillySort [y | y <- xs, y >= x]
sillySort _      = []
```

Evaluates just one element in each sublist! Almost everything is sequential!

Sorting examples

Forcing the entire spine of a list to be evaluated before returning a constructor:

```
force :: [a] -> ()
force xs = go xs `pseq` ()
    where go (_:xs) = go xs
          go [] = 1
```

Modified (for my tests):

```
force :: [a] -> ()
force xs = go xs `pseq` ()
    where go (x:xs) = x `seq` (go xs)
          go [] = 1
```

Sorting examples

A better parallel version of quicksort:

```
import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x:xs) = force grtr `par` (force lesser `pseq`
                                            (lesser ++ x:grtr))
  where lesser = parSort [y | y <- xs, y <  x]
        grtr   = parSort [y | y <- xs, y >= x]
parSort _      = []
```

Sorting examples

Fine-tuning parallelism:

```
parSort2 :: (Ord a) => Int -> [a] -> [a]
parSort2 d list@(x:xs)
  | d <= 0       = sort list
  | otherwise = force greater `par` (force lesser `pseq`
                                     (lesser ++ x:greater))
      where lesser  = parSort2 d' [y | y <- xs, y <  x]
            greater = parSort2 d' [y | y <- xs, y >= x]
            d'      = d - 1
parSort2 _ _              = []
```

On the other hand, `par` may be used quite freely, as the compiler has the freedom to not obey it.