A short introduction to concurrency and parallelism in Haskell

Matteo Pradella

matteo.pradella@polimi.it

DEIB, Politecnico di Milano

2012-2015

Explicit (classical) concurrency: Threads and MVars

As in many mainstream programming languages, parallelism can be achieved by using typical concurrency constructs, like threads and locks or monitors for shared memory access.

First, forking threads:

```
forkIO :: IO () -> IO ThreadId    -- ‘‘green’’ (or lightweight) thread
forkOS :: IO () -> IO ThreadId    -- real OS thread
```

Why are we working in the IO monad? Because most concurrent programs need to communicate with each other: this is done through shared synchronized state.

MVars: Variables with Monitors

As usual, shared memory is managed by using some form of synchronizing entities (like locks).

```
data MVar a              -- MVar can contain any data

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)

-- blocking accessors
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

```
-- non-blocking accessors
tryTakeMVar     :: MVar a -> IO (Maybe a)
tryPutMVar      :: MVar a -> a -> IO Bool

-- tests
readMVar        :: MVar a -> IO a
isEmptyMVar     :: MVar a -> IO Bool
...
```

A classical example: rendezvous

We are going to create 2 threads and a simple communication between them:

```
main = do aMVar   <- newEmptyMVar
          bMVar   <- newEmptyMVar
          endMVar <- newEmptyMVar    -- it is used to perform a ‘‘join’’
          forkOS (threadA aMVar bMVar endMVar)
          forkOS (threadB aMVar bMVar)
          c <- takeMVar endMVar      -- I’m waiting ...
          putStrLn ("ended with " ++ (show c))
```

actual code of the threads:

```
threadA valueToSendMVar valueReceivedMVar endMVar
    = do putMVar valueToSendMVar 72
         v <- takeMVar valueReceivedMVar
         putStrLn (show v)
         putMVar endMVar "the end" -- to end computation

threadB valueToReceiveMVar valueToSendMVar
    = do z <- takeMVar valueToReceiveMVar
         putMVar valueToSendMVar (10 * z)
```

let's run it:

```
*Main> main
720
ended with "the end"
```

An interlude on seq/pseq

We saw that `seq x y` is used to force the evaluation of x.
(It returns y, only if the evaluation of x terminates.)

We will use a variant called `pseq`: it has *almost* the same semantics but is useful for managing parallel computations

The difference is subtle: seq is strict in both its arguments, and the compiler may evaluate y before x (it is strict but does not enforce an order between x and y).

$pseq$ is strict only in the first argument.

Semi-Explicit Parallelism

It is the “easier” form of parallelism: we explicitly indicate to the compiler computations that can be carried out in parallel.

```
par  :: a -> b -> b      -- note: par x y = y
```

We are suggesting to compute the first argument in parallel with the second (the one whose result we are keeping).

Example: Fibonacci

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Parallel version: 1st attempt

Here is a first parallel version:

```
nfib0 0 = 0
nfib0 1 = 1
nfib0 n = par n1 (n1 + n2)
          where n1 = nfib0 (n-1)
                n2 = nfib0 (n-2)
```

(Practical note: compile with `ghc -threaded`; run with `+RTS -N`)

This version is slower than the sequential version, though!

Where are the problems?

There actually are two issues:

1. for small values of n, the overhead of threads in the parallel version outweighs the computation!

2. the evaluation of `par n1 (n1 + n2)`

The first issue is easily solved by using the sequential version for $n \leq$ of a suitable constant, e.g.:

```
nfib0 n | n <= 12 = fib n
```

Evaluation of par

The current version of $+$ in GHC evaluates first its left argument, hence $n1 + n2$ demands the value of $n1$ before starting $n2$. This blocks the potential parallelization.

Indeed, if we change the implementation like this:

```
nfib0 n = par n1 (n2 + n1)
```

we obtain roughly a 2x speedup.

A very bad idea

Clearly, this solution is bad: we should not rely on the knowledge of evaluation order of system functions – if in the next version of the compiler the change the evaluation order of parameters of the sum, our gain would be lost.

(in many functional languages the evaluation order of functions arguments is left unspecified by design)

So we need a way to specify execution order, and the usual approach is based on $pseq$:

Using pseq

```
nfib n | n <= 12 = fib n
nfib n = par n1 (pseq n2 (n1 + n2))
         where n1 = nfib (n-1)
               n2 = nfib (n-2)
```

In this case, we are forcing the evaluation of $n2$ before $n1 + n2$

In conclusion, it is quite easy to parallelize code with par and $pseq$, provided that
1) we have "expensive" computations that are independent
2) we (probably) have to specify execution order when we build up the final result.

Acknowledgments and references

Examples taken from:

Peyton Jones, Singh, *A Tutorial on Parallel and Concurrent Programming in Haskell*, 2008