Week 1
Parallel programming in Haskell

CS4012
Topics in Functional Programming
Michaelmas Term 2020
Glenn Strong <glenn.Strong@scss.tcd.ie>
Parallel Programming

• The Control.Parallel library has a number of utilities to let us signal to the runtime that there are sites of potential parallelism.

  \[
  \text{par} :: a \rightarrow b \rightarrow b \n  \]

• It doesn’t take long to persuade yourself that the only possible function that could have this type must be something equivalent to this:

  \[
  \text{par left right} = \text{right} \n  \]

• So what’s the point?
Parallel Programming

- We need to take a little digression. We know Haskell evaluates expressions *lazily*, but what does that mean.

- Here are two Haskell expressions entered at the GHCi prompt. They are very similar, but they are definitely different.

  ```haskell
  Prelude> let x = 1 + 2 :: Int
  Prelude> 3
  Prelude> x == 3
  True
  ```

- The difference is all to do with evaluation.

- We can’t see the difference in normal use:
Parallel Programming

• But there is a difference.
• “x” could be represented as “3”
• or as a lazily-unevaluated computation of “1 + 2”.

Versus
Parallel Programming

• GHCi has a debugging operation to allow you to see whether something is evaluated or not:

```
Prelude> let x = 1 + 2 :: Int
Prelude> :sprint x
x = _
Prelude> x
3
Prelude> :sprint x
x = 3
```

• This unevaluated expression that has been printed as “_” is sometimes called a “Thunk”.

Parallel Programming

• More complex examples could include the unevaluated “graphs” formed by something like this:

```plaintext
let x = 1 + 2
let y = x + x
```

• What does all this have to do with parallelism?
Parallel Programming

• Well, imagine if we have some expression graph with shared nodes.
• If the right sub tree was evaluated in parallel with the left sub tree then the result would be shared.
Parallel Programming

• So, coming back to our function:

\[ \text{par} :: a \rightarrow b \rightarrow b \]

• Semantically the expression \( \text{par} \ x \ y \) is equivalent to just \( y \), but the runtime is allowed to use it as a hint that it would be a good idea to evaluate \( x \) in parallel with \( y \).
Parallel Programming

Let’s try again. This is ex2.hs:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = par nf ( fib (n-1) + nf )
  where nf = fib (n-2)
```
Parallel Programming

![](chart.png)
Parallel Programming

- Threadscape actually looks OK at first...
Parallel Programming

• Zooming in shows the problem. We get bursts of activity followed by complete stalls.
Parallel Programming

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = par nf ( fib (n-1) + nf )
    where nf = fib (n-2)
```

What’s happening?

A new lazy task is started for nf. Nothing is demanding it’s evaluation, though...

The order that (+) evaluates it’s arguments is at the heart of this. It is *strict* in it’s left argument, so that forces fib (n-1) to be evaluated before the value of nf is demanded.
Parallel Programming

So what would happen if we swapped the order of the arguments around?

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = par nf ( nf + fib (n-2) )
  where nf = fib (n-1)
```

ex3.hs
Parallel Programming

![Graph showing time vs. cores used for different experiments (ex1, ex2, ex3).]
Parallel Programming

We shouldn’t have to have this insider knowledge of how (+) treats it’s arguments, that was way too hard.

Introducing...

\[
pseq :: a \rightarrow b \rightarrow b
\]

\textit{pseq} is like \textit{par} but it is strict in it’s first argument.
Parallel Programming

pseq causes the main task to get on with nf2

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = par nf1 (pseq nf2 (nf1 + nf2))
    where nf1 = fib (n-1)
        nf2 = fib (n-2)
```
Parallel Programming

A sidebar on syntax. In Haskell we can write any *binary* function either like this:

```
  f x y
```

or like this:

```
x `f` y
```

Sometimes it’s easier to read the program using the *infix* notation, and so I will use it from time to time:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = nf1 `par` nf2 `pseq` nf1 + nf2
  where nf1 = fib (n-1)
        nf2 = fib (n-2)
```
Parallel Programming

Spark overhead can dominate after a while. If we limit new threads to allow a better distribution of work then we can get even better performance:

```haskell
import Control.Parallel

sfib :: Integer -> Integer
sfib n | n < 2 = 1
sfib n = sib (n-1) + sib (n-2)

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

main = print $ fib 3 37
```
Parallel Programming

• You might think this is getting a little tricky to use.
• Lots of things to think about
  • Unevaluated vs Evaluated computation
  • Relative costs and sizes of computation
  • Sharing
• To explore how Haskell addresses these we need to bring in a big idea (maybe *the* big idea of this module...
Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/