Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 1
Parallel programming in Haskell

CS4012
Topics in Functional Programming

Glenn Strong <Glenn.Strong@tcd.ie>

Parallel Programming

Introduction

- Parallelism as an idea is very cool, but it can also be very hard to work with.
- Parallelism has been talked about as a good fit for Functional Programming since the earliest days of FP languages.
- Immutable data and side-effect free computation combined has a lot of promise.

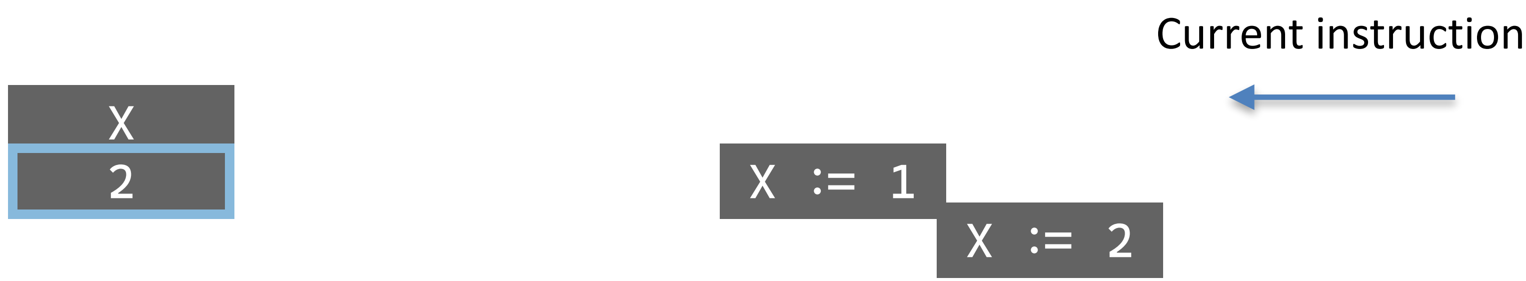
Parallel Programming

- **The various compute cores need to coordinate their activities, meaning that we (the programmers) have to think about:**
    - **Race conditions**
    - **Data dependencies**
    - **Locking (and thus locking *problems*, like deadlocks)**
- **The big problem seems to be *shared mutable data***

Parallel Programming

- **The big problem seems to be *shared mutable data.***

  When things execute in sequence it's easy to know where you are



Parallel Programming

- **But in parallel…**

Parallel Programming

- **Some terminology that we have to get right from the outset:**
  - *Parallelism* **is using multiple compute elements (e.g. more than one CPU core) to perform a computation.**
  - *Concurrency* **is the use of more than one thread of program control at a time (which may, or may not, involve more than one processor).**

Parallel Programming

An example: Fibonacci numbers

A small example to help us explore…

```
fib :: Integer → Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
main = print $ fib 37
```

Parallel Programming

Compiler flags

GHC has some built-in options to enable:

- **Multicore**
- **Performance analysis (profiling)**
- **Configuring the runtime system**

```
ghc -threaded -rtsopts -eventlog ex1.hs
```

Link with the threaded runtime

enable passing options to the runtime

link a runtime that can record traces

Parallel Programming

Runtime flags

- **Running it a few times with different options to the Run Time System will give us some data about how multiple cores speed up the process**

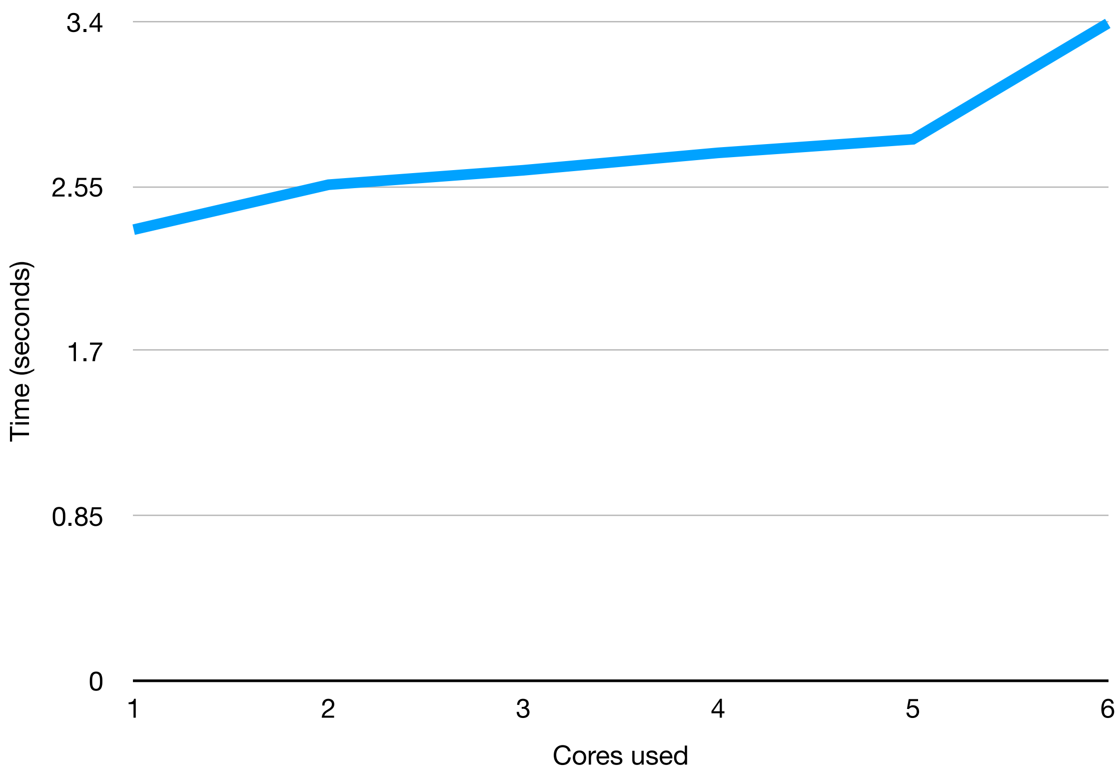
use this many threads (default : 1)

```
time ./ex1 +RTS -N1
time ./ex1 +RTS -N2
time ./ex1 +RTS -N3
… and so on…
```

pass the following options to the runtime

Parallel Programming

An example: Fibonacci numbers. Timings



Parallel Programming

- This is very discouraging.
- The program did not get any faster when using multiple cores
  - In fact, it got a little bit slower!
- To find out what has gone wrong we will use the ThreadScope profiling tool.
- If you pass the "-l" flag to the runtime it will dump a log of various interesting events to a file ("ex1.eventlog").
- This binary file is not the easiest thing to read on it's own, we need

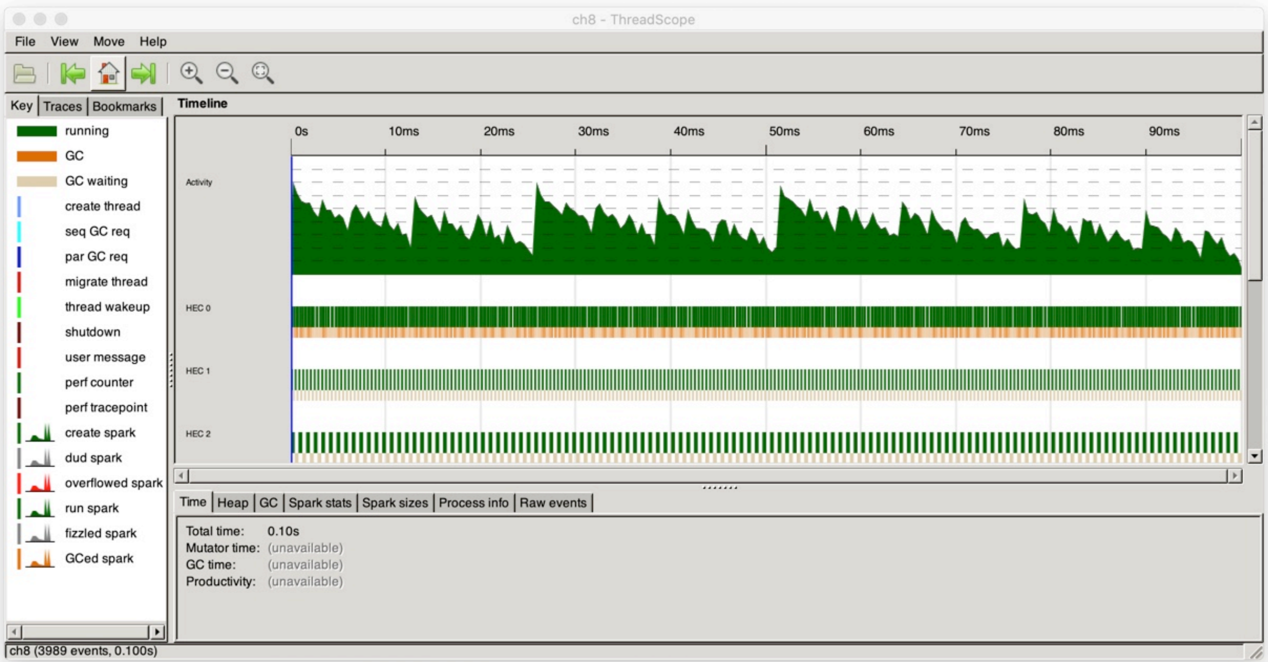
Parallel Programming

ThreadScope

- ThreadScope is a profiling and analysis tool for parallel Haskell.
- It can be a little bit awkward to install from source
- I advise using a pre-built binary if you can. Instructions are on the Haskell Wiki and linked on Blackboard.
    - You need to install the Haskell GTK bindings
    - Then the ThreadScope binary
    - I will make a VM available for this if you need it.

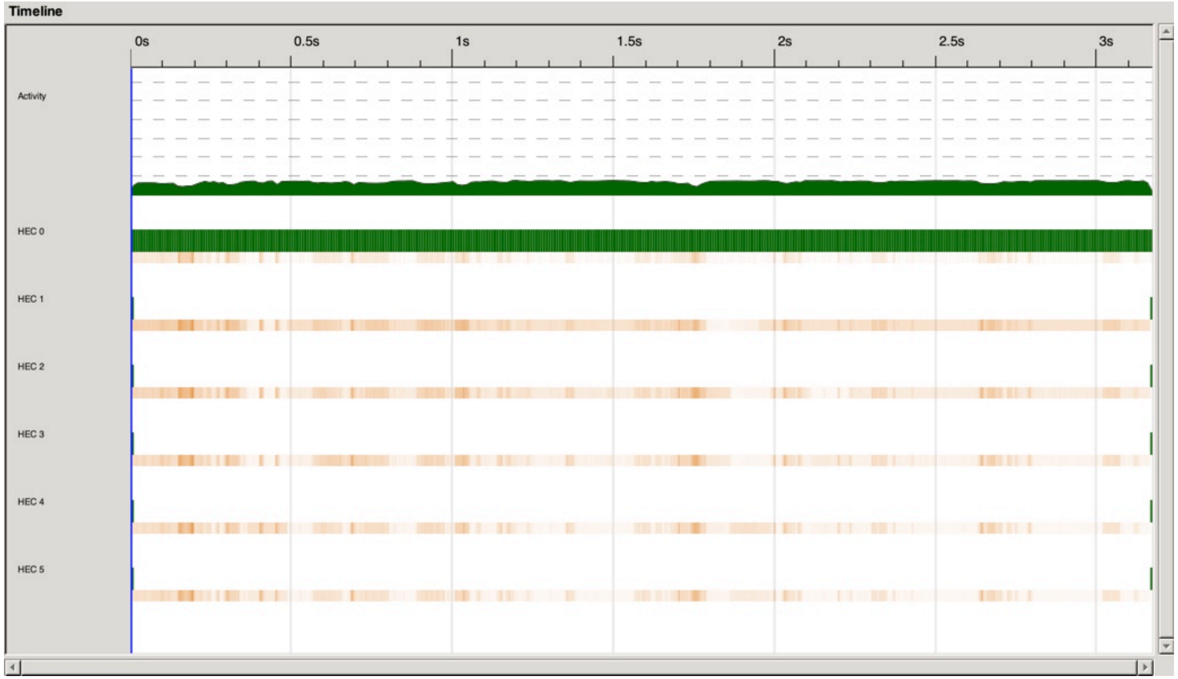
Parallel Programming

- **The Threadscope UI**



Parallel Programming

- **What do the ThreadScope results look like for our ‘fib’ program?**



Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

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

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 1
Parallel programming in Haskell

CS4012
Topics in Functional Programming

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.**

```
par :: a → b → b
```

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

```
par left right = right
```

  - **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.**

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

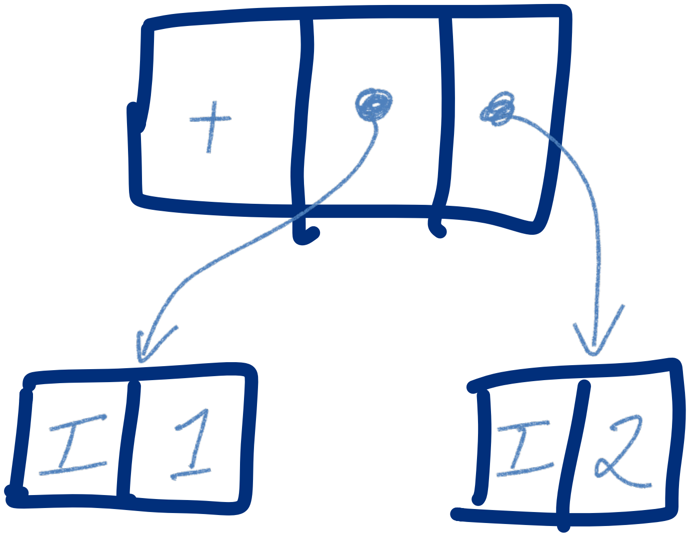
- **The difference is all to do with evaluation.**
- **We can't see the difference in normal use:**

```
Prelude> x == 3
True
```

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:

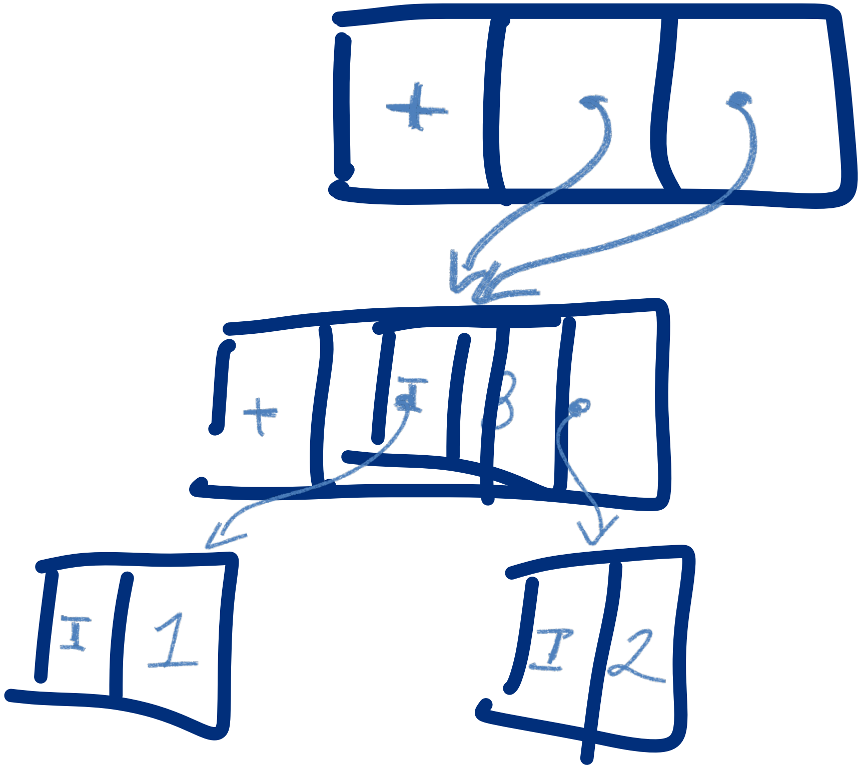
```
Prelude> let x = 1 + 2 :: Int
Prelude> :sprint x
x = _
Prelude> x
3
Prelude> :sprint x
x = 3
```

- 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:**

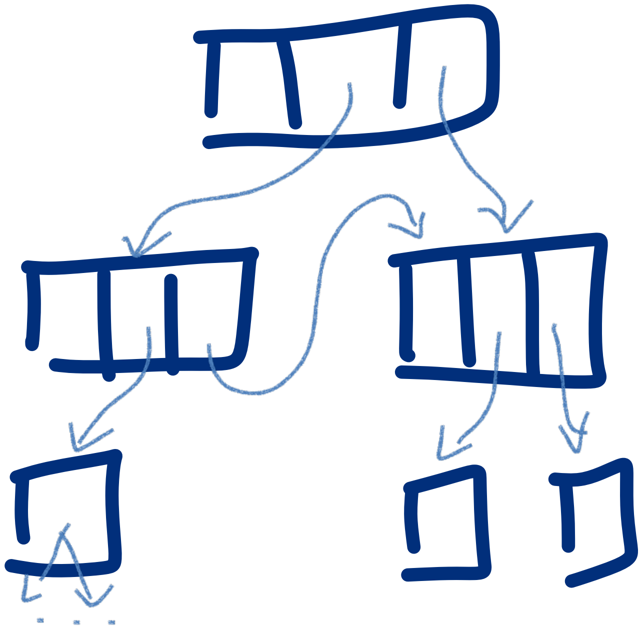
```
let x = 1 + 2
let y = x + x
```



- **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:

```
par :: a → b → b
```

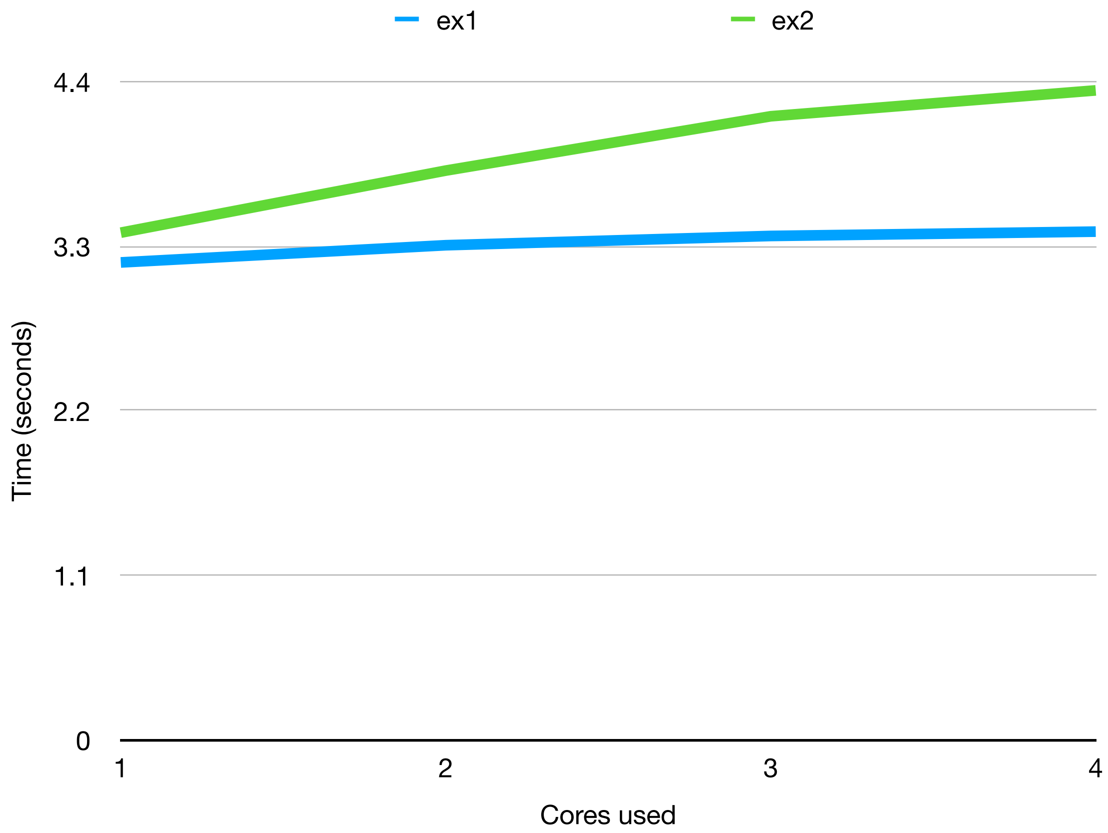
- Semantically the expression `par x y` is equivalent to just `y`, but the runtime is allowed to use it as a hint that it would be a good idea to evaluate `x` in parallel with `y`

Parallel Programming

Let’s try again. This is ex2.hs:

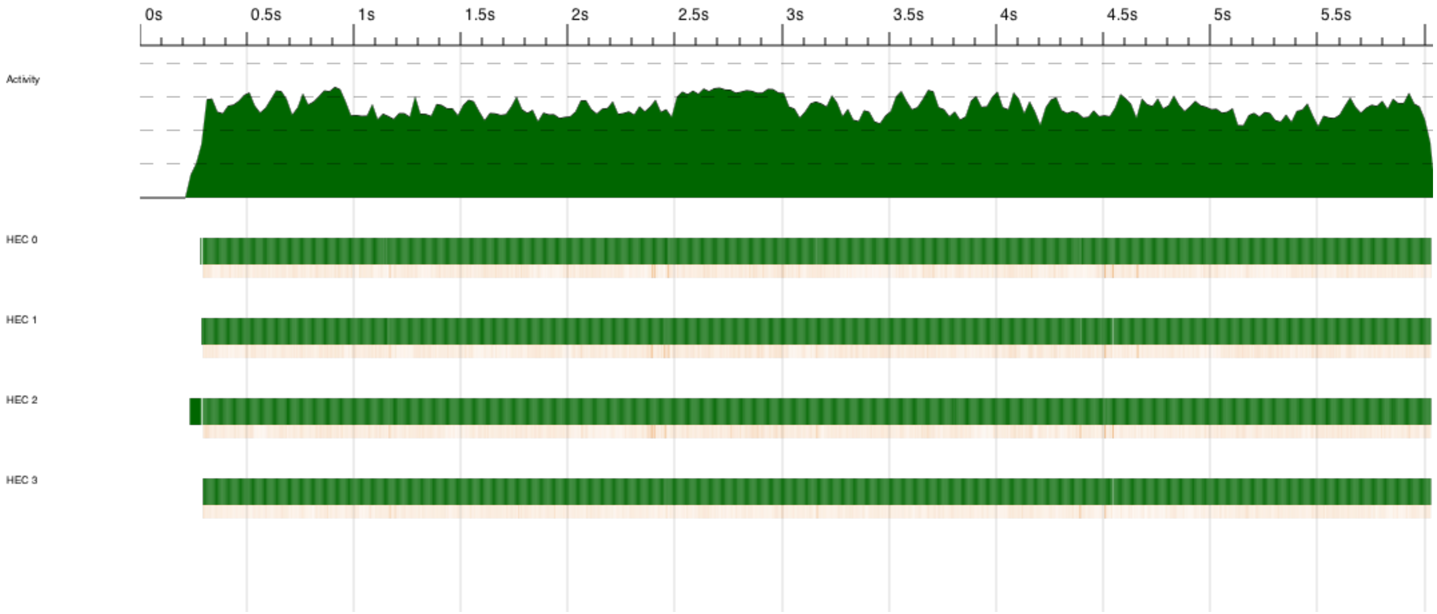
```
fib :: Integer → Integer
fib 0 = 0
fib 1 = 1
fib n = par nf ( fib (n-1) + nf )
          where nf = fib (n-2)
```

Parallel Programming



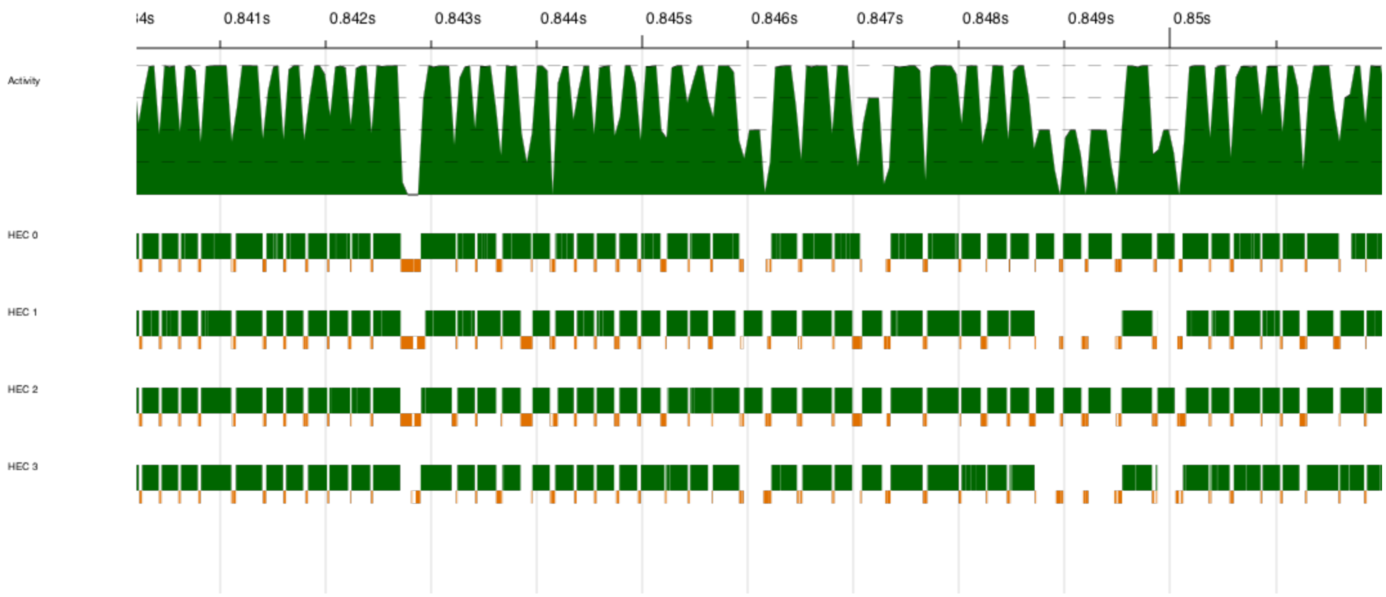
Parallel Programming

- **Threadscope 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.

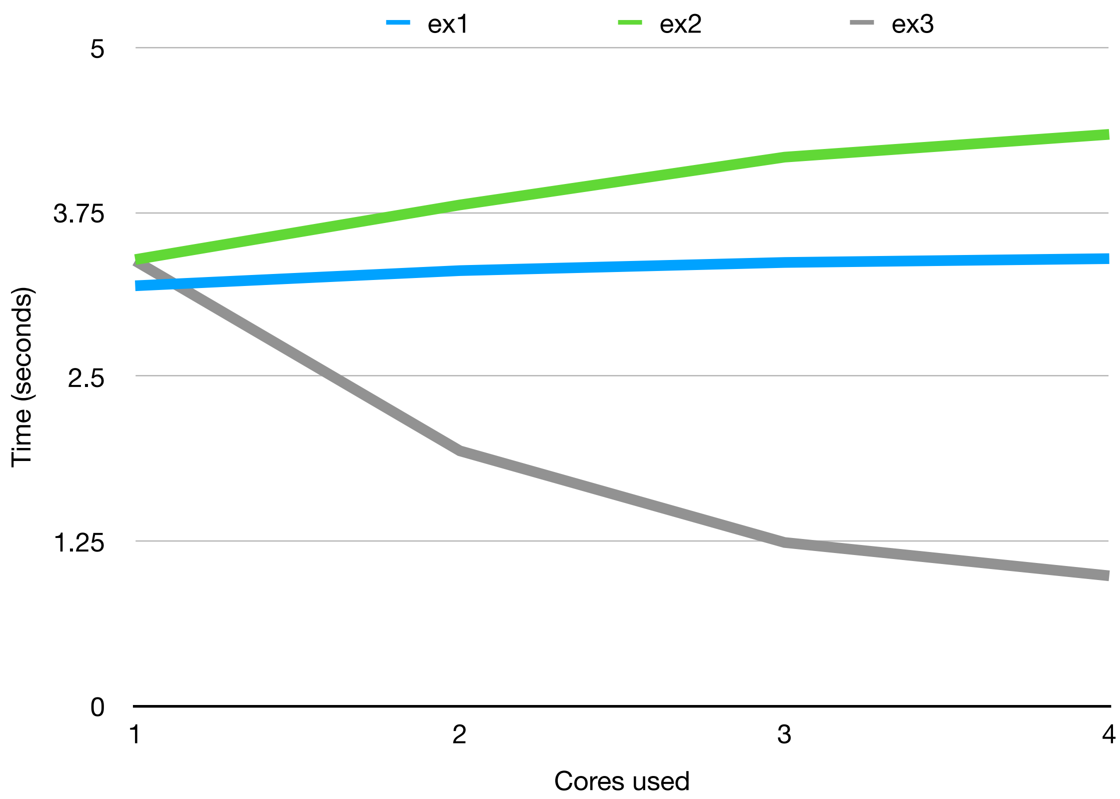
Parallel Programming

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

```
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



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 → b → b
```

pseq is like par but it is strict in it's first argument.

Parallel Programming

pseq causes the main task to get on with nf2

```
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:

```
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…)

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

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