CS4012 Topics in Functional Programming

Glenn Strong

Parallel Programming in Haskell

- Parallelism has been talked about in functional programming circles since the earliest days
- Immutable data and side-effect free computation has a lot of promise

A small example

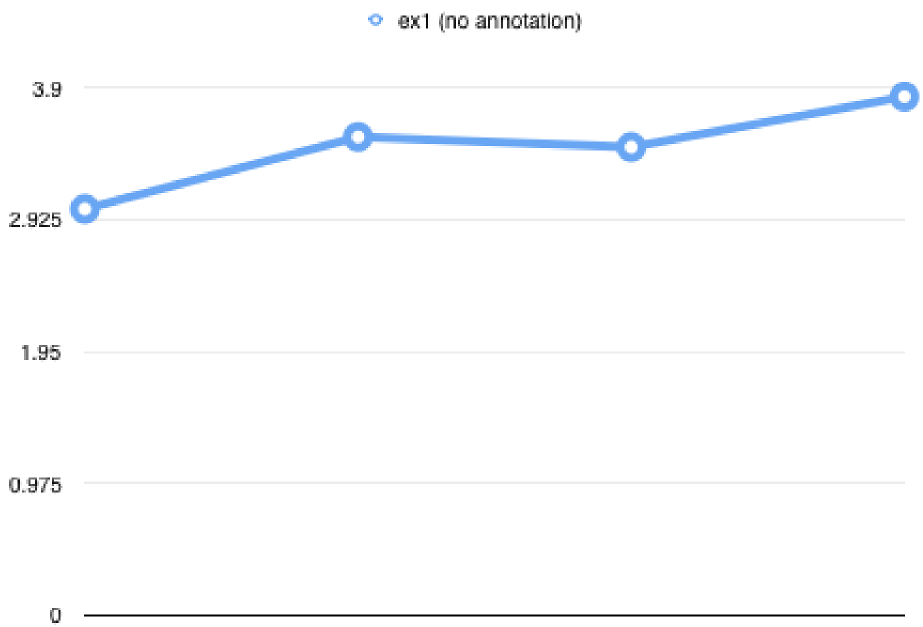
```haskell
fib :: Integer -> Integer
fib n | n < 2 = 1
fib n = fib (n-1) + fib (n-2)

main = print $ fib 37
```

Compile:

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

Sample runs:

```
$ ./ex1
$ ./ex1 +RTS -N1       # 1 Core
$ ./ex1 +RTS -N4       # 4 Cores
$ ./ex1 +RTS -N4 -ls   # 4 Cores + event log
```

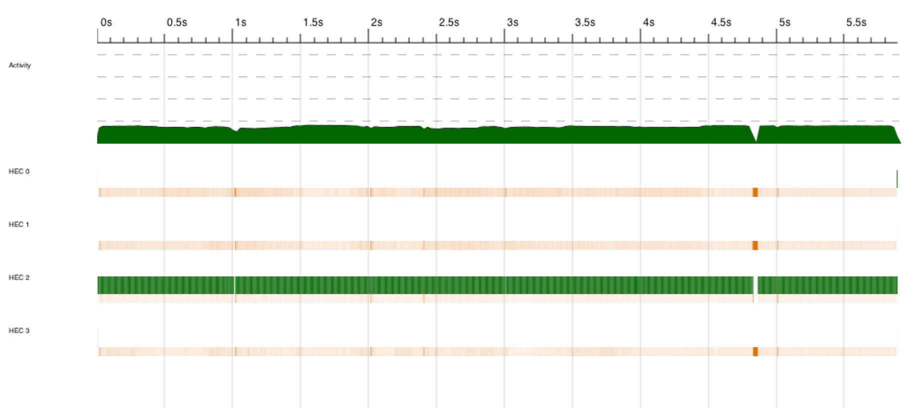


Event logs

We can look at the event log using the ThreadScope tool

- demo time!
- ThreadScope can be a bit of a pain to install
- Use a prebuild binary if you can

ThreadScope main view



Results!

That's no so good. Let's try *telling* the run-time where some parallelism is possible.

"Spark" parallelism

The `par` operation (from the `Control.Parallel` library) allows us to signal sites of potential parallelism.

```
par :: a -> b -> b
```

Semantically `par x y` is equivalent to just `y`, but the runtime is allowed to use it as a hint.

Next try

```
import Control.Parallel

fib :: Integer -> Integer
fib n | n < 2 = 1
fib n = par nf ( fib (n-1) + nf )
          where nf = fib (n-2)

main = print $ fib 37
```

Timing

It actually got slower!

Threadscope

It looks OK at first…

Threadscope

Zooming in shows lots of threads stalling…

What's happening?

```
fib n = par nf ( fib (n-1) + nf )
          where nf = fib (n-2)
```

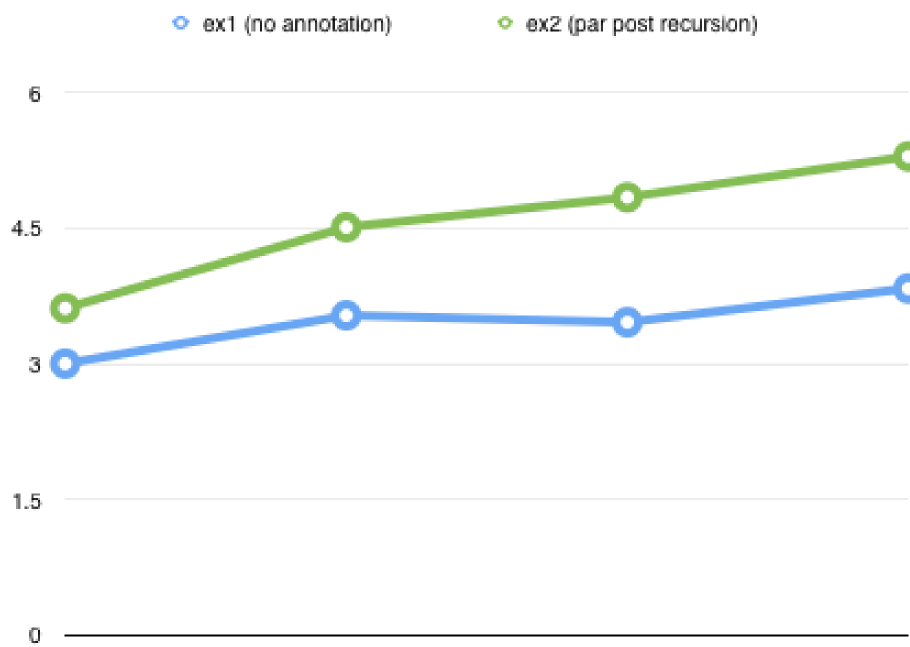


Figure 1: Runtime vs HEC's

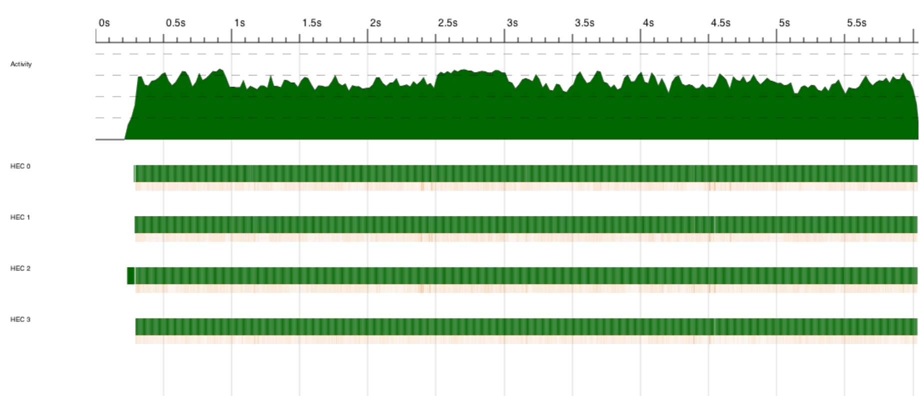


Figure 2: ThreadScope(example 1, 4 cores)

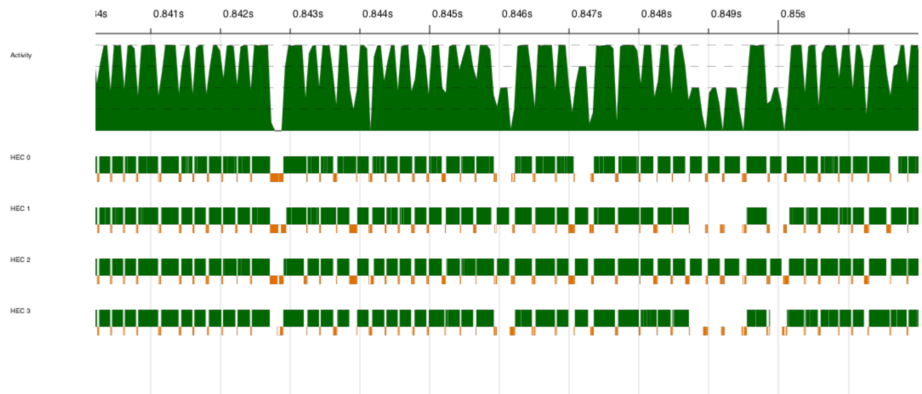


Figure 3: ThreadScope(example 1, 4 cores)

There is a new task for each `nf`; one starts and then blocks on the other right away!

The order that `(+)` evaluates it's arguments is at the heart of this.

Taking over a bit

So if we just swap around the arguments would that be enough?

```
fib n = par nf (nf + fib (n-2) )
          where nf = fib (n-1)
```

That's crazy!

A better way...

We shouldn't have to care about how `(+)` treats its' arguments.

Introducing:

```
pseq :: a -> b -> b
```

Using pseq

Evaluate `x` *before* `y`, returning `y`.

```
fib n | n < 2 = 1
fib n = par nf1 (pseq nf2 (nf1 + nf2))
        where nf1 = fib (n-1)
              nf2 = fib (n-2)
```

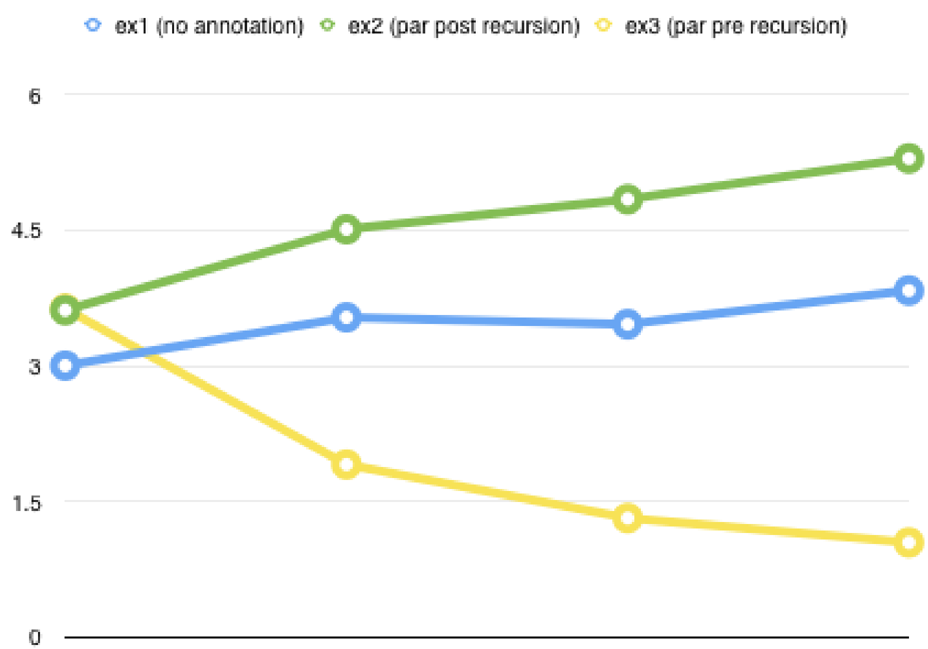


Figure 4: Runtime vs HEC's

Further tweaking

Spark overhead can dominate after a while. Limit new threads to allow a more even distribution of work?

```haskell
import Control.Parallel

-- Sequential version of fib, when we want to avoid parallelism
sfib :: Integer -> Integer
sfib n | n < 2 = 1
sfib n = sfib (n-1) + sfib (n-2)

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

main = print $ fib 3 37
```

Going further

- This is all still a bit tricky to use
- Lots of things to think about
  - Evaluated vs. Unevaluated computation
  - relative costs and sizes of computation
  - Sharing
- To explore how Haskell addresses these we need to take a digression…