The par monad.
Used for deterministic parallel computation
The tutorial build mainly on a paper on Par monad from (Marlow et al. 2010) were the Par monad is introduced.
Par and pseq have been used as a kind of basic for general-purpose deterministic parallelism and also the framework Evaluation Strategies by (Trinder et al. 1998) (Marlow et al. 2010).
The user needs to understand operational properties and makes par difficult to use.

We have par, pseq :: a → b → b
were par creates a spark, parallelism occurs, for its first arguments and then evaluates its second argument.

Problem with par:
a) pass unevaluated computation to par, because we dont evaluate it, only put it inside a spark-pool.
b) we have to require the spark to be used somewere wich is required inside the computation, like a while-loop.
c) the result most be shared to the rest of the program.

A and C are were the parallelism occurs, and c is more a job for the garbage-collector.
The rules looks simple, but it requires some kind of reasoning before we can use par. To make sure we dont violate any of these rules, par monad is introduced.

The par monad
It was introduced to be able to reasoning in deterministic way and to not need to learn new syntax for it, because its a library.

```
newtype Par a

instance Functor Par
instance Applicative Par
instance Monad Par

runPar :: Par a → a
```

runPar is were the computations from the Par monad can be extracted.
The guarantee that runPar produces a deterministic result for any given computation in Par monad, the result does not have any side effects and no I/O.
We see it from the fact that the type of runPar is Par a → a.
The downside is that runPar is expensive in its computation and would normaly only be used during small tasks but on large data.

```
Fork :: Par () → Par ()
```

Fork function makes so we can create parallel tasks.
It works like this: the input to fork executes concurrently with the current computation.
Its works similar to a tree.
We need to communicate in the tree, so the result in fork can be used to the parent in the tree.
There is were Ivars comes in.

```
data Ivar a – instance Eq

new :: Par (Ivar a)
get :: Ivar a → Par a
put :: NFData a => Ivar a → a → Par ()
```

Ivar is similar to an array and is inmutable with operations get and put.
Put assigns a value and can only be used once. If done two times, we will get run-time error.
Get waits until Ivar has been assigned a value, then returns the value.
One reason we create a monad is because this type of data structure wont work in purely functional data structure.

NFData (normal form data) context on put makes put fully strict. Everything trough Ivar will be fully evaluated.
Normal process is to thread fork into several children (in the tree) and then collects the results.

```
spawn :: NFData a => Par a → Par (Ivar a)
spawn p = do
        I ← new
                fork (do x ← p; put I x)
                return I
```

Ivar in here is called a future, because its the value that are going to be completed at another computation at later date.

We could create spawn to a list. But a better option is to combine spawn with map.

```
ParMapM :: NFData b  => (a → Par b) → [a] → Par [b]
parMapM f xs = do
        b ← mapM (spawn. f) xs
        mapM get b
```

Its similar to a map but creates a child process for each of the element and then waits until everything is completed.

If we look at an example of a sorting algorithm, quicksort.
A sequentional one would look like this:

```
quicksort :: Ord a => [a] → [a}
quicksort [] = []
quicksort (x:xs) = quicksort [y | y ← xs, y < x] ++ [x] ++
                        quicksort [y | y ← xs, y >= x]
```

We already see how it calls the function recursivity two times, which is a good place to start with parallelism. The idea here is to just call these functions in a tree-like structure, so we wait until all the function calls its done. Its done with spawn. So its like the same as the sequential quicksort but added spawn and the get functions.

```
parQsort :: (NFData a, Ord a) => [a] -> [a]
parQsort [] = []
parQsort (x:xs) = runPar $ do
  less <- spawnP . parQsort $ filter (< x) xs
  bigger <- spawnP . parQsort $ filter (>= x) xs
  [l,g] <- mapM get [less, bigger]
  return $ l ++ [x] ++ g
```

The problem with parallelism is it cost to create processes. Therefore, we need to stop parallelise at some depth – a thershold. The problem is to find this threshold and we do trial-and-error right now.

Think the parallelism is like a tree, so every time it spawn a process, we move down the tree. It means that we could simply add a number everytime we create a process which decreases once for every spawned processes.

```
dParQsort :: (NFData a, Ord a) => Int -> [a] -> [a]
dParQsort d [] = [] -- return
dParQsort 0 xs = qSort xs -- return
dParQsort d (x:xs) = runPar $ do
  less <- spawnP $ dParQsort (d-1) $ filter (< x) xs
  bigger <- spawnP $ dParQsort (d-1) $ filter (>= x) xs
  [l,g] <- mapM get [less, bigger]
  return $ l ++ [x] ++ g
```

Problem with parallelising quicksort is the use of pivot element and the data might not be splitted even and might be slower than sequential code.

Benchmarking with an array with 100 000 integers. We take the mean of some computations and get:
sequential: 446 ms
parallel: 3.789 s
parallel with depth 4: 343 ms

There might be problem with other processes interfering the benchmark during runtime. But it shows how the parallel version is slower, mainly because it creates processes for every time it runs. The parallel with depth tries to make use of every core on the computer but still not to much, which explains its improvement from the parallelising.
From figure 1, we see it uses almost 2 cores, but dosnt use the other cores.
Figure 2 shows the sequential code, and clearly this only uses one core and not to fully extent.

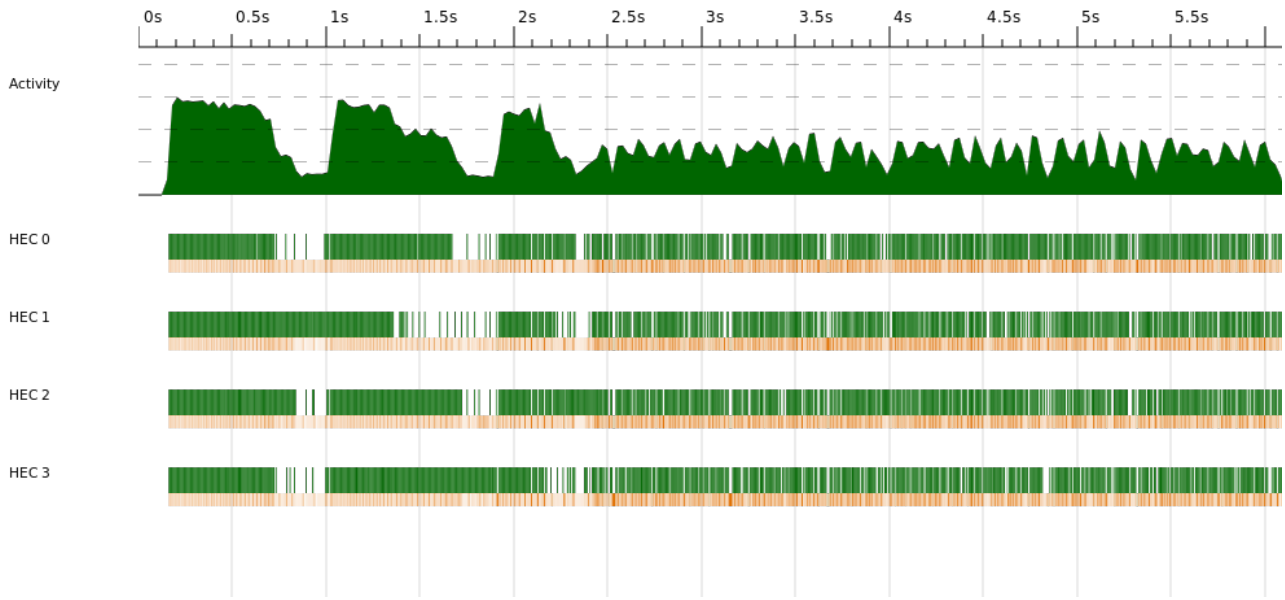

Figure 1, threadscope with parallelise with depth on 4 cores.

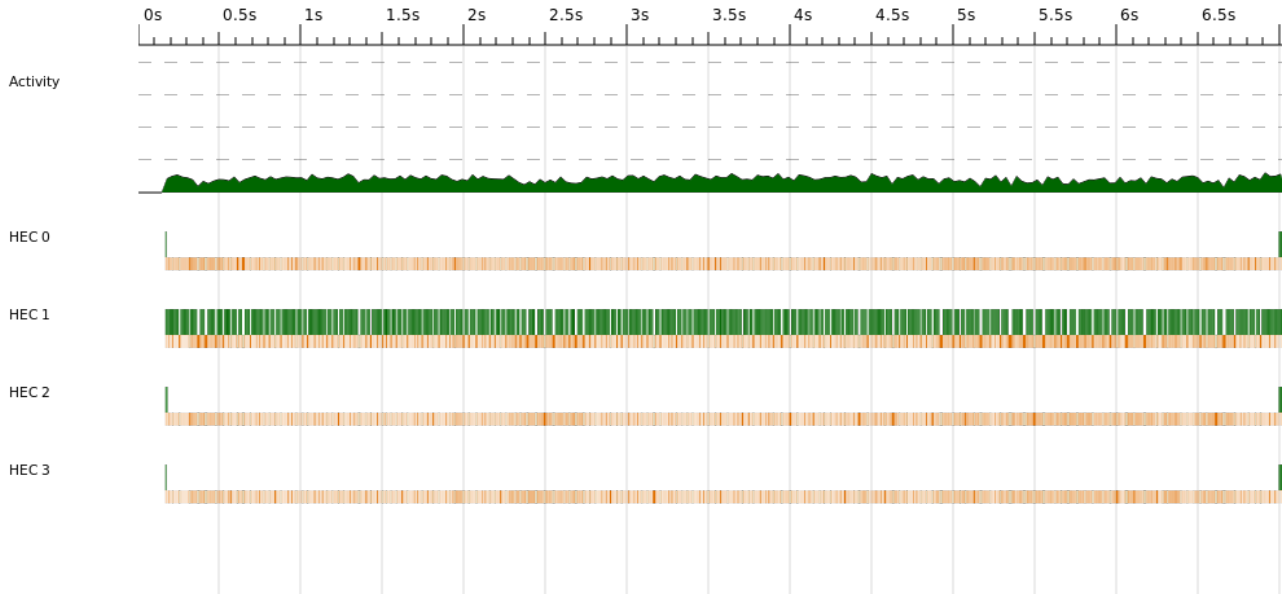

Figure 1. Sequential quicksort.

References

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad,and Phil Trinder. Seq no more: better strategies for parallel Haskell.InProceedings of the third ACM Haskell symposium on Haskell, pages91–102, 2010.

PW Trinder, K Hammond, H-W Loidl, and SL Peyton Jones. Algorithm +strategy = parallelism. 8:23–60, January 1998