Par monad tutorial

Johan Swetzén
swetzen@student.chalmers.se

Sebastian Lagerman
barafinkel@gmail.com

May 11, 2014

Introduction

The `Par` monad provides parallelism by modeling the flow of data. Instead of explicitly stating what should be run in parallel, the dependencies of the computations decide how many processes there will be. The parallel computations are written, as one would expect, in the Par monad and then the results can be extracted using `runPar`.

```
runPar :: Par a -> a
```

To start computations in parallel, there is a function called `fork`.

```
fork :: Par () -> Par ()
```

As you can see, the `fork` function doesn't return any data, so instead there is a data type for this, called `IVar`. An `Ivar` is a container for a value computed in parallel and there are a few functions provided for using it.

```
new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

The `new` function creates an `IVar`, `put` fills it with a value and `get` takes it out once it has been fully computed.

Factorial

Let's start off by parallelising a simple factorial function. First, a sequential version might look like this.

```
fac :: Integer -> Integer
fac n = product [1..n]
```

Say we want to split this into two parts in parallel. Then we need to declare two new `IVars`, `fork` the left and right computation and then join them together. Since this is such a simple example, the dependencies look like this:

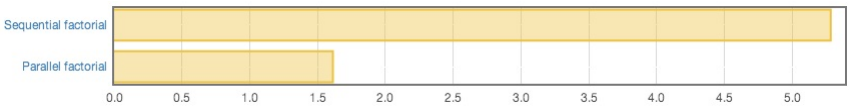


Figure 1: The results of running the fac-function and facP-function on a two core machine

```
    Left  Right
      \   /
      return

fac :: Integer -> Integer
fac n = product [1..n]

facP :: Integer -> Integer
facP n = runPar $ do
  left <- new
  right <- new
  fork $ put left $ product [1..mid]
  fork $ put right $ product [(mid+1)..n]
  l <- get left
  r <- get right
  return $ l * r
    where mid = n `div` 2
```

All this `IVar` declaring can look a bit messy, so there are some functions to help us avoid both `new` and `put`.

```
spawn :: NFData a => Par a -> Par (IVar a)
spawnP :: NFData a => a -> Par (IVar a)
```

`spawn` is like `fork`, but it creates the `IVar` for you. `spawnP` is defined as `spawn . return`, so we can call it with a pure function directly. This simplifies our code a bit.

```
facP' :: Integer -> Integer
facP' n = runPar $ do
  left <- spawnP $ product [1..mid]
  right <- spawnP $ product [(mid+1)..n]
  [l,r] <- mapM get [left,right]
  return $ l * r
    where mid = n `div` 2
```

Let us have a closer look at the work flow of the two cores during facP-function (see figure 2). What we can see is that there is some garbage collection due to the usage of arrays and possibly that the cores aren't using their full power which could be because there are only two tasks. The last part when

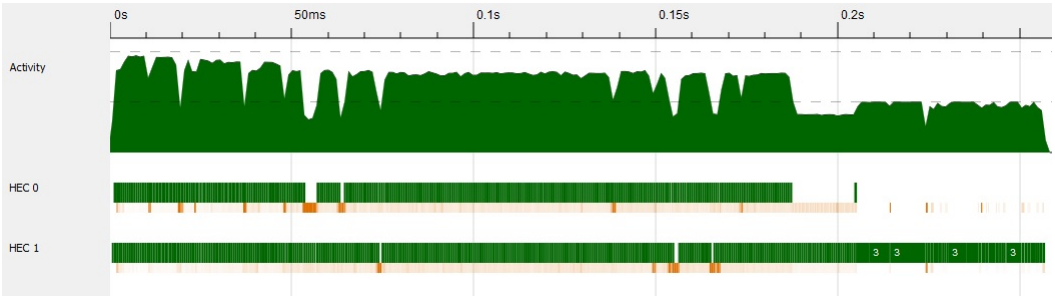


Figure 2: How the two cores worked during a running of facP-function.

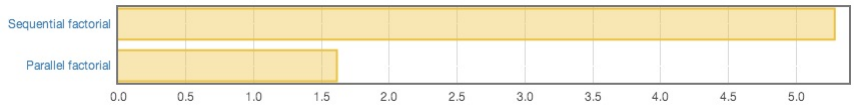


Figure 3: The results of running the seq fac-function and facP-function on a two core machine

only one core is active is most likely the combination stage of the two arrays we get from the two tasks.

Quicksort

Now when we have a better understanding of how `spawn` and `get` works let us discuss a problem with a bit more parallelism. We will see the importance of making sure there are enough parallel tasks available, but not too many too small ones. In this example we will implement a quicksort function. It's a divide & conquer algorithm which is called recursively on the elements with lesser and greater values as seen here:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort lesser ++ [x] ++ qsort greater
  where
    lesser = filter (< x) xs
    greater = filter (>= x) xs
```

Now as we have learned there is an easy way to parallize this function which is to spawn two tasks for the lesser and greater functions used in the `where`. Something like this:

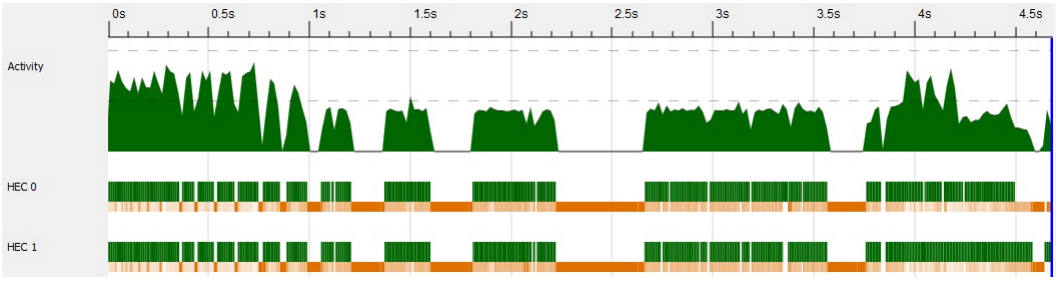
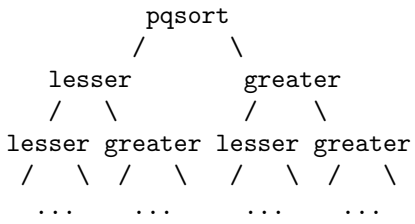


Figure 1: How the two cores worked during a running of pqsort-function.

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

Let us look how the two cores are working while running pqsort-function (see figure 1). Something clearly seen in this figure is that half the work these core are performing are garbage collection. Partially because we are using arrays and most like because too maney sparks are created to be practical.

This however would create two new tasks for each call of pqsort until the list is empty, resulting in $\mathcal{O}(logn)$ tasks on a list with n elements. The dependencie graph looks like this:



Spawning that many tasks will be problematic since it takes some work to start up a task. The end result is that most of the parallel processes are so small that the overhead of spawning the tasks is far greater than the actual task we want to perform. This can be seen in figure 3, where the parallel version is slower than the sequential one. The way to get around this problem is to implement a depth which tells the program how many recursive calls we want it to do until it should do the last calls sequentially. Which in short means that we decide how many tasks we want and their sizes. One easy way to implement this is to add an `Int` parameter to the function that decreases for each step and once it reaches zero it will call the sequential function:

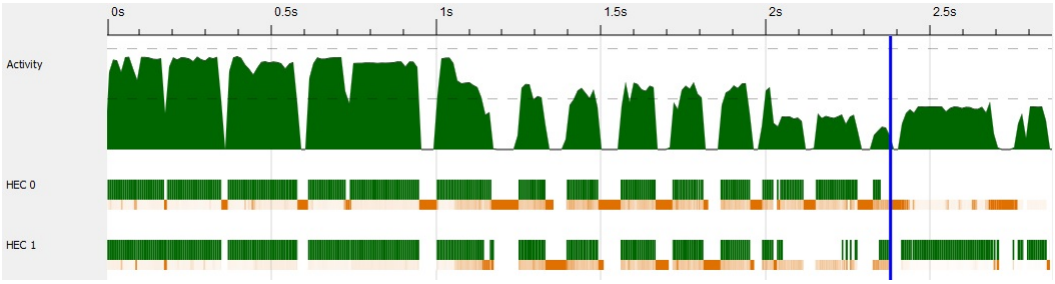


Figure 2: How the two cores worked during a running of qsort2-function.

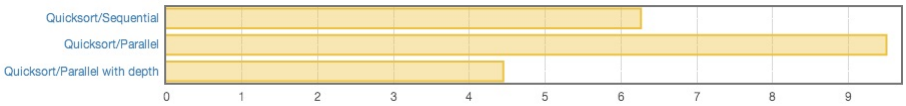


Figure 3: The results of running the seq qsort-function and pqsort-function on a two core machine. Note that the Quicksort/Parallel is sorting a five times smaller list than the other two to make the others visible at all.

```
pqsort2 :: (NFData a, Ord a) => Int -> [a] -> Par [a]
pqsort2 _ [] = return []
pqsort2 0 xs = return $ qsort xs
pqsort2 d (x:xs) = do
  lesser <- spawn $ pqsort2 (d-1) $ filter (< x) xs
  greater <- spawn $ pqsort2 (d-1) $ filter (>= x) xs
  l <- get lesser
  g <- get greater
  return $ l ++ [x] ++ g
```

Once we are done with parallizing our function then we can have a look at how much two cores are working (see figure 2). We see there are a fair amount of garbage collection, which are most likely caused by the usage of arrays. Otherwise the two cores work almost all through the program.