A tutorial on Parallel Strategies in Haskell

by
Oscar Andersson
Yanling Jin

This tutorial assumes some previous Haskell experience, including some understanding of Monads and preferably having seen par and pseq before.

1. Introduction

There are surprisingly many ways to do deterministic parallelism in Haskell. One of the more complete libraries is called “Strategies”, which provides built-in support for controlling granularity by chunking, evaluation control and, of course, “sparking”.

A spark is the atomic unit of work in the GHC runtime system. A spark points to a thunk suitable for evaluation. The RTS scheduler will select sparks from the spark pool (which is a circular buffer) when it has no runnable threads. Still, the computation will not stall when the RTS hasn’t had any time to evaluate its sparks. The regular program flow will not hold just because there are unevaluated sparks left -- it will evaluate them on-the-fly.

Since Haskell is lazy, we know that operations like mapping over a list doesn’t necessarily evaluate the whole list, if not carefully referenced in such a manner. This tutorial will deal with how to control evaluation in Haskell. The Strategies libraries make it easy to express what level of evaluation you expect from your input and when you want it.

Compiling and how to profile GHC programs

Don’t forget that you need need to compile your programs with, at least, the -threaded flag. This compiles the runtime system with the ability to spawn OS threads. This is a good idea if you want anything more than an academic exercise.

The GHC runtime system is capable of outputting logging information about its internal events. Coupled with a visualization tool like ThreadScope, we get a decent profiling tool. Make sure to compile your programs using the the -eventlog flag.

To be able to set important runtime options, compile with the -rtsopts flag.

Depending on what compilation options you prefer, compiling a program may look something like

```
ghc --make -threaded -eventlog -rtsopts -O2 program.hs
```

2. A first example

As always with Haskell, we’ll learn by doing some Fibonacci.

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

-- The Eval Monad version
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
            nf1 <- rpar (qfib (n-1))
            nf2 <- rpar (qfib (n-2))
            return (nf1 + nf2 + 1)

-- The strategy version
sfib :: Integer -> Integer
sfib n | n < 2 = 1
sfib n = -- nf1 + nf2 + 1 `using` strat
         withStrategy strat nf1 + nf2 + 1
    where nf1 = nfib (n - 1)
          nf2 = nfib (n - 2)
          strat v = do rpar nf1; rseq nf2; return v
```

There's a close relationship between the Eval monad and Strategies as we can see in the Haddock documentation. Strategies step in as a higher-level abstraction to ease many of the problems associated with regulating granularity and forcing evaluation manually.

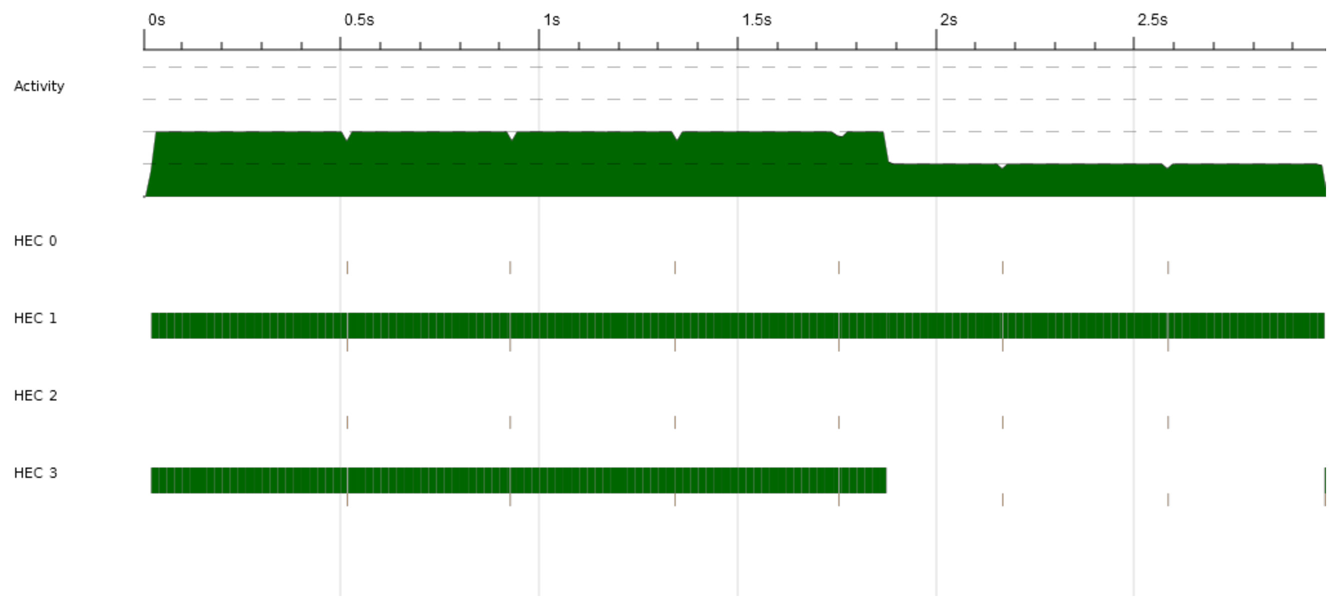
```haskell
type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
x `using` s = runEval (s x)
withStrategy :: Strategy a -> a -> a
withStrategy s x = runEval (s x)
```

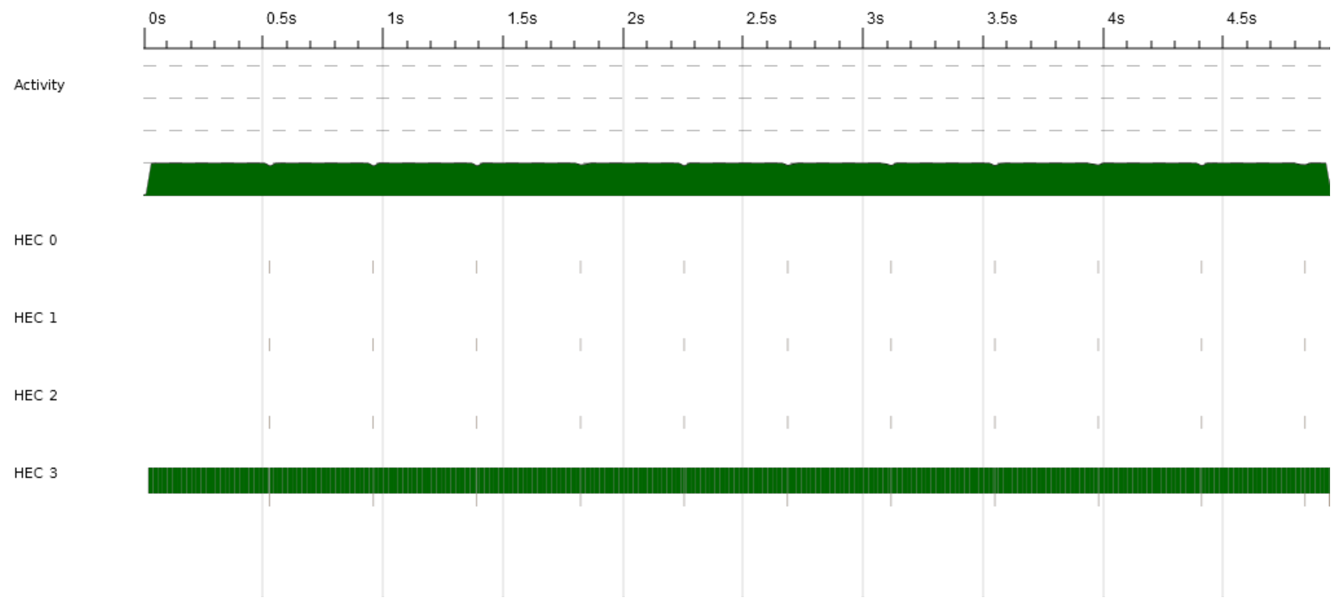
"using" and "withStrategies" are logically equivalent. They're in the library to provide syntactic sugar.

Strategies can in many ways be seen as an extension of the Eval monad, with some syntactic sugar and evaluation order control.

Connecting to the code examples above, it's very hard to reason theoretically about how sfib will execute. The strategy rpar provides no guarantees about evaluation, it is merely a hint to the RTS that it might be a good idea to work on these thunks when there are no runnable threads for the RTS to schedule. When running sfib on a 4-core system, we expect the spark to be scheduled, due to the lack of competing threads. When trying to profile this program using GHC eventlogs and ThreadScope, this is what running sfib 40 looks like:

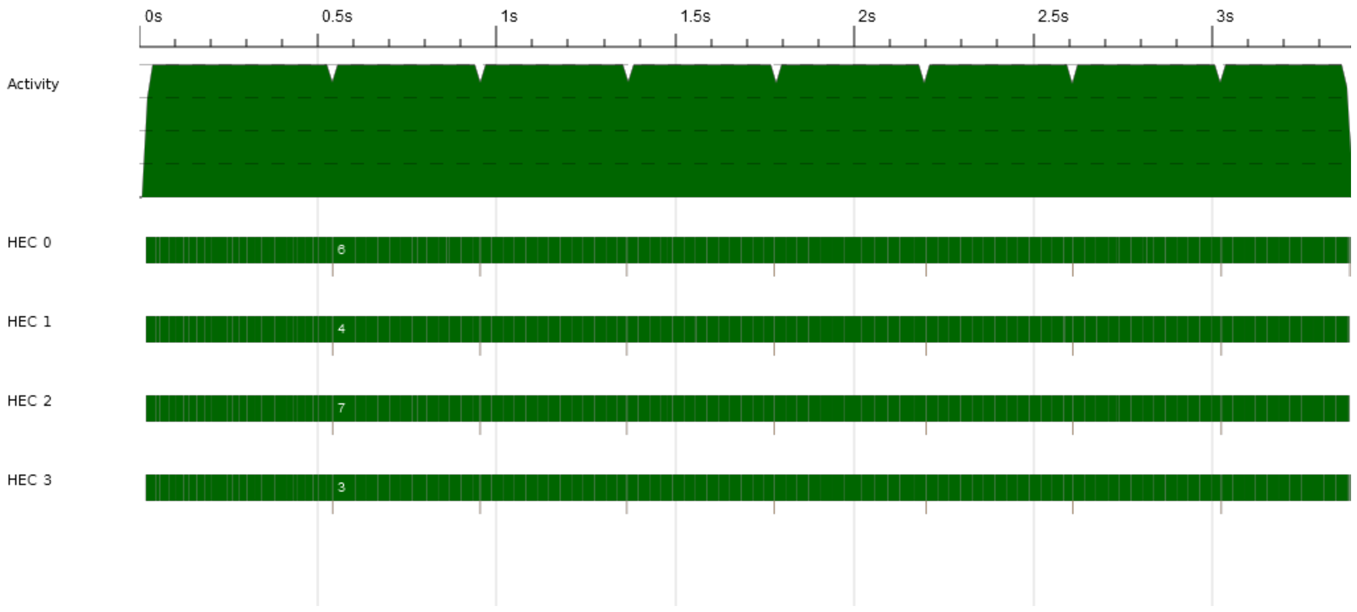


It seems like we achieved good utilization of two cores for a significant period of time. But was it useful work? Let's compare the wall clock time to running nfib 40, our sequential variant and building block of sfib.

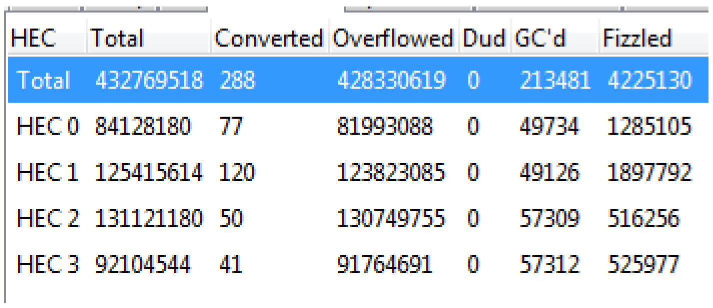


Look at the time axis -- sfib is a huge improvement to nfib! So what happened in the sfib example? We didn't spark off a bunch of sparks and do work stealing, but the thunks evaluated in main thread were apparently accessible to the spark thread and vice versa.

Looking at the qfib version, it seems like it will allow for even more parallelized execution. It calls itself recursively, spawning more sparks at every call. Let's run qfib 40, and feed the eventlog through ThreadScope.



It looks like we're using the machine to its full extent, but it's performing worse than sfib, which, at best, only utilized two cores on the machine. Let's check out the spark stats from the eventlog.



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	432769518	288	428330619	0	213481	4225130
HEC 0	84128180	77	81993088	0	49734	1285105
HEC 1	125415614	120	123823085	0	49126	1897792
HEC 2	131121180	50	130749755	0	57309	516256
HEC 3	92104544	41	91764691	0	57312	525977

That's an insane amount of sparks! Let's go through the meaning of these terms. **Converted** means that the spark was actually evaluated by an available core and turned into useful work. **Overflowed** refers to sparks discarded due to the spark pool being full. **GC'd** refers to sparks being garbage collected. **Fizzled** were the sparks already evaluated by the normal program flow.

Sparking off work in the GHC runtime system is associated with much lower overhead than spawning kernel threads, or forking a process, but there is still overhead. We want to find a balance between qfib, which is sparking way too many units, and sfib, being able to utilize a maximum of two cores. Also, sparking off tasks with rpar and hoping for stuff to evaluate didn't serve us very well here. Strategies will remedy this to a certain extent as we will see, and also, Strategies incorporate control of the granularity as part of the library.

3.1 Scan revisited

A useful family of functions that very gracefully lets themselves be parallelized are the family of scan functions. Below is a parallel implementation using Strategies.

```
scanP :: (Num a, NFData a) => Int -> (a -> a -> a) -> [a] -> [a]
scanP d f list = concat reducedList
  where
    scanList = map (scanl1 f) (chunk d list) `using` parList rdeepseq
    reducedList = reduce f scanList
    strat v = do rpar reducedList; return v

reduce :: (a-> a-> a) -> [[a]] -> [[a]]
reduce f [] = []
reduce f (x:[]) = [x]
reduce f (x:(y:xs)) = x : reduce f (map (f $ last x) y : xs)

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = as : chunk n bs where (as,bs) = splitAt n xs
```

```
 scanList = map (scanl1 f) (chunk d list) `using` parList rdeepseq
```

Here we see the outermost building block of Strategies, the infixed "using" function, evaluating an expression using a strategy.

```
x `using` s = runEval (s x)

parList :: Strategy a -> Strategy [a]
```

"parList" is arguably one of the most straightforward strategies in the library. It evaluates each element of a list in parallel as sparks according to a given strategy. In this example, the granularity control built in to Strategies is not used, as we want to retain control over the reduce step. If you're not so concerned about the reduce step of your algorithm, go ahead and use the "chunking" built-in to Strategies. It would look something like this:

```
 scanList = map (scanl1 f) (chunk d list) `using` parListChunk 4
rdeepseq
```

3.2 Reasoning about evaluation in Haskell

We'll now turn our attention to the strategy rdeepseq. What does it mean, and what is the NFData typeclass defined in the type signature of scanP?

The NFData typeclass introduces the constraint that all arguments of type a must be evaluated to normal form, which means that there are no more beta reductions to apply to the expression. For most practical purposes, one can say that the expression is evaluated when it is in normal form, that there are no unevaluated thunks left.

The "rdeepseq" is the function actively enforcing this evaluation to normal form, and makes Haskell behave more like a strict programming language. "rpar" is the strategy of leaving the evaluation to the spark pool, as we've seen before. "rseq" is a weaker evaluation strategy, only evaluating the list to weak head normal form, for lists meaning that the head will be evaluated.

Note that rdeepseq as a strategy can be useful even in sequential Haskell programming.

```
evalList :: Strategy a -> Strategy [a]
```

is the sequential counterpart of parList. Stringing this Strategy together with rdeepseq can be a powerful tool in constructing efficient algorithms.

It was mentioned earlier that the laziness of Haskell requires more consideration of the programmer when it comes to evaluation. It is not customary for Haskell to fully evaluate lists unless that information is required as part of some requested computation.

3.3 Other Strategies

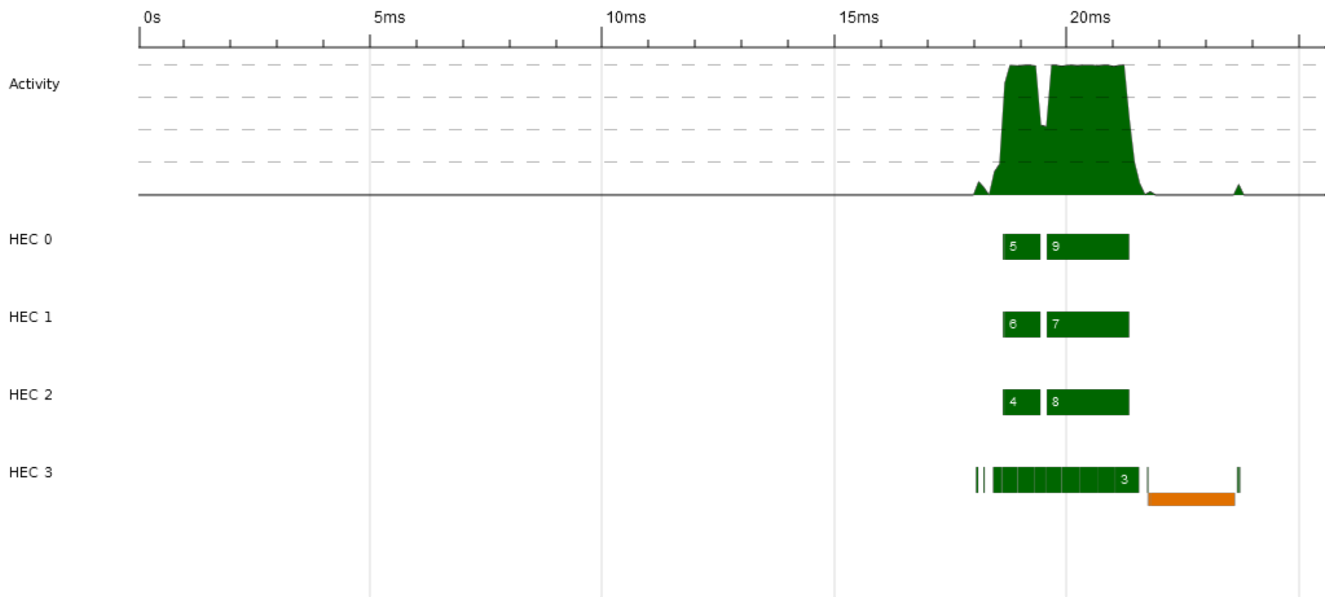
```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

parListChunk, in contrast to parList, does not spark evaluation of all list elements individually, but rather divides the list into a fixed amount of chunks, which will also be the number of sparked tasks. The reduce step is built-in.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

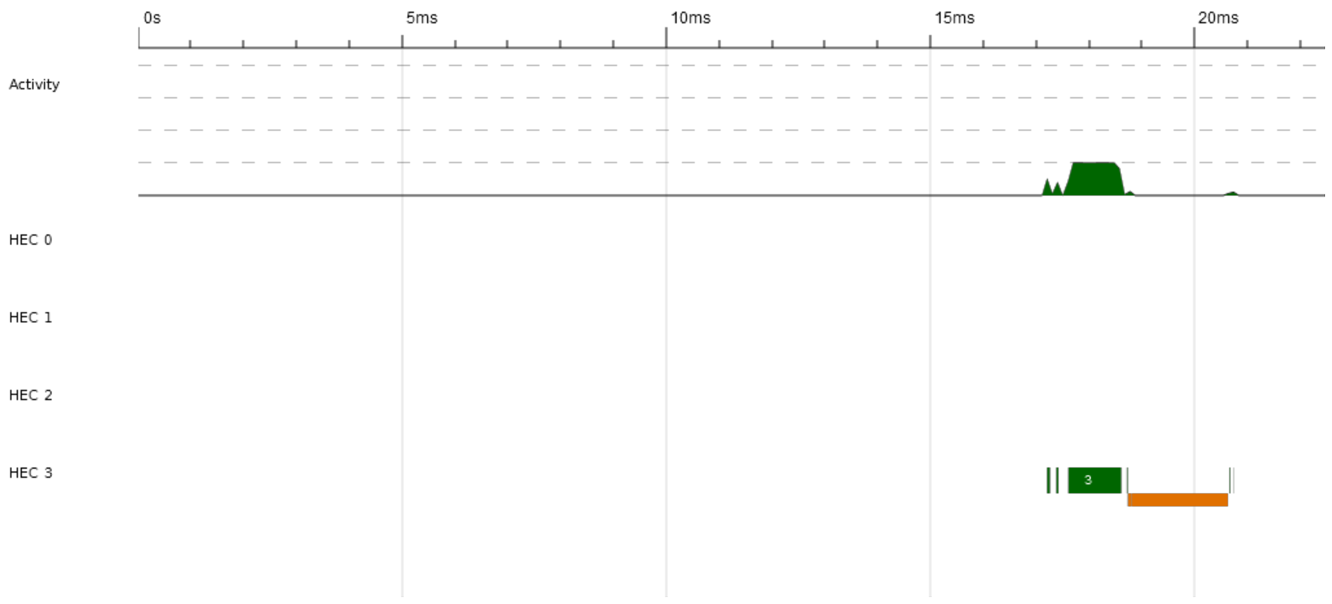
It should not shock you terribly that there is a supplied Strategy for mapping in parallel. In ideal cases, you can get multicore speedups by doing little more than mechanical code substitution, like this:

```
-- |Add 20 to each integer in the list.
addMap :: [Integer]
addMap = map (+20) l
addMap = parMap rdeepseq (+20) l
```



A visualized eventlog of running the parallelized addMap over 30 000 elements

Bingo! The conversion from the sequential version to the parallel version didn't take any effort at all. This can be done mechanically for every Haskell program in existence.



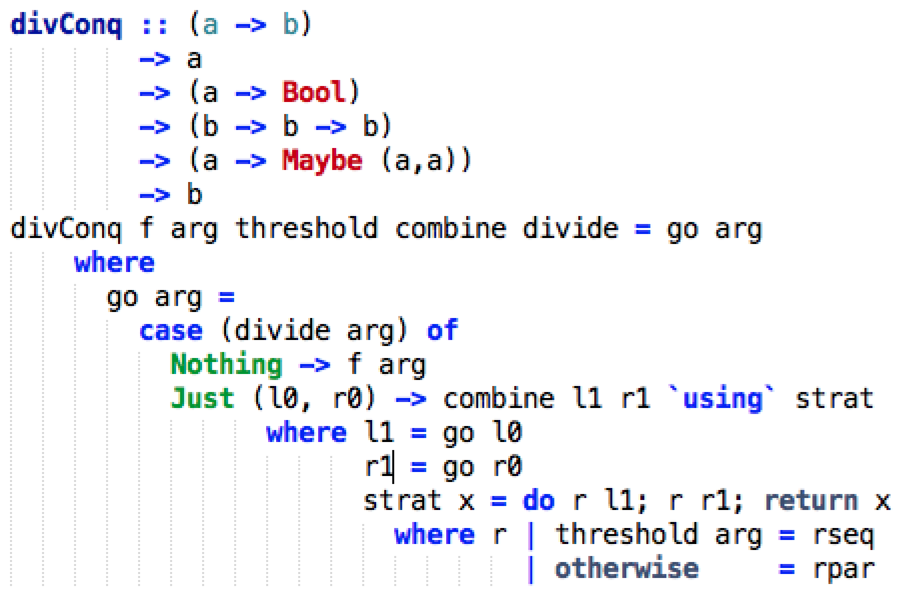
A visualized eventlog of running the sequential addMap over 30 000 elements

But it wouldn't be a good idea. This goes to show that in order to achieve good results, Haskellers need a good understanding of the underlying components, from the runtime system to evaluation order.

4. Skeletons

Strategies are well suited for writing skeletons in most cases, since one of the major strengths of strategies is the ability to provide abstraction over parallel computation patterns. ParListChunk in Section 3 and ParMap can be seen as skeletons.

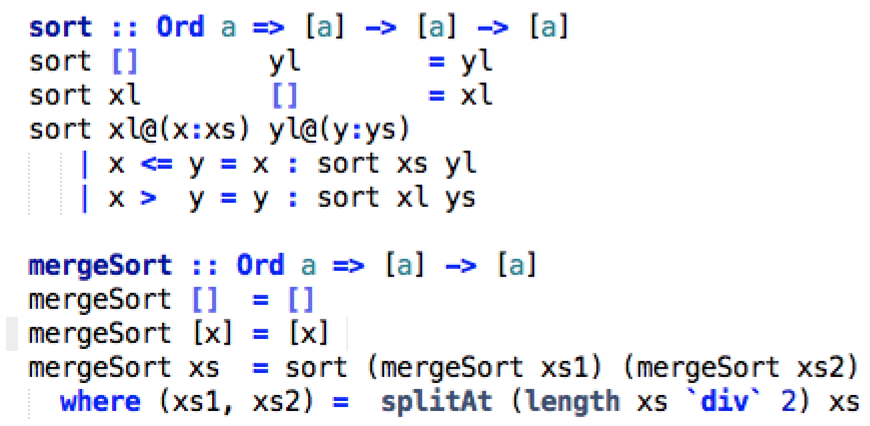
Divide-and-conquer is a typical example of using strategies to separate algorithm from parallelism. It is often expressed as follows:



```haskell
divConq :: (a -> b)
    -> a
    -> (a -> Bool)
    -> (b -> b -> b)
    -> (a -> Maybe (a,a))
    -> b
divConq f arg threshold combine divide = go arg
  where
   go arg =
    case (divide arg) of
     Nothing -> f arg
     Just (l0, r0) -> combine l1 r1 `using` strat
        where l1 = go l0
           r1 = go r0
           strat x = do r l1; r r1; return x
            where r | threshold arg = rseq
                | otherwise     = rpar
```

DivConq *divides* a non-trivial problem into subproblems, and applying the same schema *f* to each subproblem. The final solution is a *combination* of the solutions of the subproblems. "strat" encodes the spark of subcomponents l1 and r1 as well as how they should be evaluated. The *threshold* controls the depth of the parallelism.

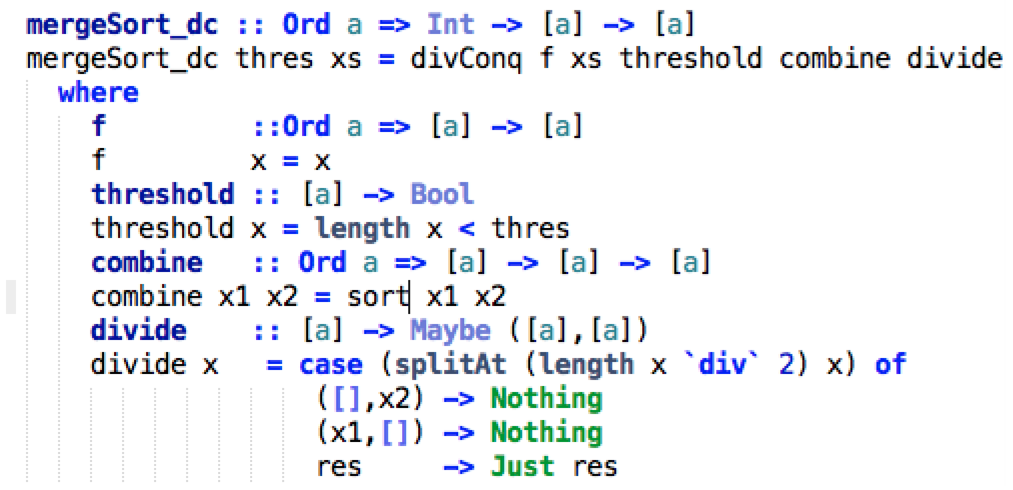
We will use mergesort as an example to demonstrate the use of divide-and-conquer skeletons. Sort combines two lists and performs sort using the standard Haskell function. A sequential version of mergesort can be implemented as following.



```haskell
sort :: Ord a => [a] -> [a] -> [a]
sort []          yl          = yl
sort xl          []          = xl
sort xl@(x:xs) yl@(y:ys)
    | x <= y = x : sort xs yl
    | x >  y = y : sort xl ys

mergeSort :: Ord a => [a] -> [a]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = sort (mergeSort xs1) (mergeSort xs2)
  where (xs1, xs2) =  splitAt (length xs `div` 2) xs
```

With this Divide-and-Conquer skeleton, we can implement a parallel version of mergesort. It entirely separates the algorithm from the parallelism. There is no code in this implementation which is dedicated to concurrency.



```haskell
mergeSort_dc :: Ord a => Int -> [a] -> [a]
mergeSort_dc thres xs = divConq f xs threshold combine divide
  where
    f          ::Ord a => [a] -> [a]
    f          x = x
    threshold :: [a] -> Bool
    threshold x = length x < thres
    combine   :: Ord a => [a] -> [a] -> [a]
    combine x1 x2 = sort x1 x2
    divide    :: [a] -> Maybe ([a],[a])
    divide x    = case (splitAt (length x `div` 2) x) of
                    ([],x2) -> Nothing
                    (x1,[]) -> Nothing
                    res     -> Just res
```

5. Summary

Deterministic parallelism in Haskell shows great promise, but there is no free lunch. Sprinkling parallel constructs everywhere can hamper performance if not used carefully, as we have seen. We have however shown that there are other higher-level frameworks available to the programmer apart from OS threads, which in our opinion are more suitable for describing algorithms.

6. More information

For more information, visit the Strategies webpage at
http://hackage.haskell.org/package/parallel-2.2.0.1