APL Report - Parallel Programming in Haskell with Strategies

s1139243

November 7, 2014

Abstract

This report gives an overview of parallel programming in Haskell, with a particular focus on Strategies from the Control.Parallel.Strategies module. A brief description and simple usage cases are given, informing about the background and general method of using Strategies. This is tested in practice on an example of Monte Carlo sampling to determine the area of Scotland from an image of its map. The example demonstrates how strategies can be manipulated to act on different data structures. The resultant benchmark shows a visible, but not ideal gain in execution speed, for reasons specific to the example. It is concluded that Strategies are a flexible way to specify parallelism on a very high level, but devising custom ones, in Haskell's case, requires some knowledge of the language's internals.

Parallelism in Haskell - precautions

[2] The purity of Haskell has received much praise for making its programs yield well to parallelisation. However, another feature of Haskell, lazy evaluation, makes it slightly trickier. To get around it, it is important to know how lazy evaluation treats expressions. Any expression from declaration until evaluation lives as a thunk in memory and is not executed. Only when its value is requested does it execute and turn into the concrete value. Lazy evaluation gives a performance benefit by calculating only the values that are needed, at the same time ruling out speculative execution, necessary for parallelism.

Another concept in Haskell is the weak head normal form, which is an intermediate state of evaluation for an expression. In this state, only the structure of the result is known, but not its component values. For a list, it would either be known that it is empty list or a constructed list, or, in some cases, the length might be known, but the elements would remain thunks. Unfortunately, this also makes parallelisation a bit more difficult as we need to explicitly tell Haskell to evaluate an expression completely, to the normal form.

Basic primitives pseq and par

To fit parallelism into this scheme, we need a way to force eager evaluation on certain thunks before their value is queried. They should also be made to run in different threads. In Haskell there are special primitives that allow this: *par* and *pseq*, each taking two expressions to evaluate in parallel or in sequence, respectively.

These primitives can be used as infix operators.

```
operation1 `par` operation2 `par` ... `par` operationN
```

This denotes that all operations can be executed in parallel. However, this way of specifying parallelism is low level and can get tedious on more complicated structures. An additional pitfall is that seq only evaluates the argument to weak head normal form, so programmers would have to force deep evaluation explicitly. This cumbersome usage was the rationale for creating Strategies.

Strategies

On the battlefield, a strategy is the plan that needs to be executed in order to achieve some aim. This plan is general and can be applied to multiple situations that have some similar aspects. Likewise, a Haskell Strategy is a separate description of parallelism that can be reused on different computations, such as traversing lists or trees, applying divide and conquer algorithms and other parallelisable entities. Essentially, a strategy works on the pattern, not the data itself.

In Haskell Strategies are identity objects (i.e. returning the same value that was passed to them), built on top of the aforementioned par and pseq instructions. The current version of *Control.Parallel* (3.2.0.4) implements the *better* strategies introduced in the *Seq no more* paper[1]. Some example strategies are listed below, but more are available in the *Control.Parallel.Strategies* module [3]

```
rseq :: Strategy a
```

The simplest strategy, which simply evaluates its argument sequentially to weak head normal form.

```
rdeepseq :: NFData a => Strategy a
```

Sequentially evaluates its argument to normal form. Obviates the need to specify deep evaluation in the argument's body, the strategy itself does that.

```
rpar :: a -> Eval a
```

Creates a spark for its argument, meaning that it can be evaluated in parallel.

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

Evaluates a list in parallel, creating a spark for each element. Useful when elements are computationally heavy.

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

Divides a list into chunks and assigns a spark to each. Could be used for splitting a long list of relatively simple calculations across multiple cores.

Simply calling the above functions on expressions does not trigger their evaluation, but returns a *Strategy* object in the *Eval* monad. Expressing parallelism in a monad is a way of imposing order on execution and combining different strategies. It can be applied to a thunk using the *withStrategy* function or with the '*using*' infix operator. A simple usage example follows.

```
1 let l = [x*x | x <- [1..10000]]
2 l `using` parList
```

Here, l is a list of thunks representing squared numbers from 1 to 10000. On line 2 it is getting evaluated with the parList strategy. To specify parallel computations in a more manageable fashion, one can do it in the *Eval* monad directly, and run it using *runEval*. Doing this, it is possible to control the flow of execution, which in Haskell is otherwise not predetermined.

```
1 runEval ( do
2   a <- rpar $ sum [x*x | x <- [1..1000]]
3   b <- rseq $ sum [x+x | x <- [1..1000]]
4   return a + b )
```

This example does two sum operations in parallel - first a is sparked, then b is run sequentially, causing the a spark to start running as well. In fact, every application of a strategy takes place in *Eval*, which can be verified from the definition of *using*:

```
1 using :: a -> Strategy a -> a
2 x `using` strat = runEval (strat x)
```

A practical example

Given an image (1000x1513) of a map of Scotland [9], we would like to determine its approximate land area to give tourists an idea of how much there is to see. Testing whether a point on the map belongs to Scotland or not is easy, so one approach is to go through each pixel, counting those belonging to land. However, checking each of the 1.5 million points of the image separately is a waste of resources if we are just interested in the approximation.

Better suited for this purpose is a Monte Carlo method, which in this case would involve testing a smaller number of random samples of pixels from the map to get the proportion of land in it. Provided we know the rectangular area of the whole map, deriving just the area of Scotland is trivial using the calculated proportion. Given that there are no dependencies between samples, this method can be conceptually parallelised and will yield useful for trying out Haskell's Strategies.

To implement and benchmark the method and its parallelisation, a test program was made, which outputs just the ratio of land to water of the map.

The source code, as well as a Linux 64bit binary can be downloaded from `http://olafs.eu/r/misc/strat.zip`
Compile by running *make* or

```
ghc -O2 strat.hs -rtsopts -threaded
```

-rtsopts - enables the +RTS switch on the binary, used for turning on multiple cores and measuring of the execution statistics

-threaded - enables multithreading

-O2 - general optimizations for speed

Run using:

```
./strat <samples> <parallel> +RTS -N -s
```

samples - number of test samples to generate

parallel - True to enable parallelism, False to run sequentially

+RTS -N - uses maximum available cores (use -N2 for 2 cores, -N4 for 4 cores, etc)

+RTS -s - displays statistics, which are explained below

The code uses the *JuicyPixels*[4] package to read the image and *Control.Parallel.Strategies*, both installable using *cabal*. The full code listing and its explanation is shown below.

```
1  import System.Environment
2  import System.Random
3  import Codec.Picture
4  import Data.List
5  import Control.Parallel.Strategies
6  import Control.Concurrent
7
8  --Generates a list of n random integers x, where 0 <= x < max
9  randomList :: Int -> Int -> StdGen -> [Int]
10 randomList n max = map (\x -> (abs x) `rem` max) . take (n + 1) .
       unfoldr (Just . random)
11
12 --Tests whether a given coordinate at index i from xCoords and
       yCoords in img is land
13 coordIsLand :: [Int] -> [Int] -> Image PixelRGB8 -> Int -> Bool
14 coordIsLand xCoords yCoords img i = land $ pixelAt img (xCoords !!
       i) (yCoords !! i)
15     where
16     land (PixelRGB8 r g b) = (r == 248) && (g == 159) && (b == 27)
17
18 --Chunked application of a function on a list, like map
19 --Takes in Int - how many chunks to split in
20 parMapChunk :: Strategy b -> Int -> (a -> b) -> [a] -> [b]
21 parMapChunk strat chunkSize f = withStrategy (parListChunk
       chunkSize strat) . map f
22
23 --Main program
24 main :: IO()
25 main = do
26     [argument1, argument2] <- getArgs
27     let samples = read argument1
```

```
28      let parallel = read argument2
29      seed <- newStdGen
30      imgFile <- readImage "scotlandmap.png"
31      case imgFile of
32          Right (ImageRGB8 img@(Image width height _)) ->
33              let xCoords = randomList samples width seed
34                  yCoords = randomList samples height seed
35                  mapper = if parallel then parMapChunk rdeepseq
    chunkSize else map
36                       where chunkSize = ceiling $ fromIntegral
    samples / 4
37                  landList = mapper (coordIsLand xCoords yCoords img)
     [1..samples]
38                  hits = length . filter id --counts ones in a list
39                  positiveSamples = hits landList
40              in
41                  print $ fromIntegral(positiveSamples) /
    fromIntegral(samples)
42          _ -> print "Unexpected image format"
```

Code walkthrough

Line	Description
9-10	The randomList function, which generates a list of random numbers. Used for the lists of x and y coordinates separately.
13-14	A function that turns a coordinate from an image into a boolean indicating whether there is land. This is done by a color test on line 16.
20-21	parMapChunk, a helper function which applies the passed function to every element of the list, using the parListChunk strategy
24	Program entry point
26-28	Reading in the command line arguments
29	Setting the seed for the random number generator used in the random coordinate lists
30	Reading in the image using JuicyPixels
31-32	Matching image format with the one we are prepared to work with
33-34	Generating the coordinate lists
35	mapper is the map function applied to the list - this is selected to be sequential or parallel, depending on the 2nd command line argument
36	chunkSize is set to split number of samples equally across the number of cores (hardcoded to 4)
37	Applies mapper to the list of indices [1..samples]
38-39	Auxiliary functions used below
41	Calculating the ratio itself and printing it
42	Error in case the we cannot match the image format on line 32

Parallelising the example

The process that is being parallelised is the mapping from the index list *[1..samples]* to booelans indicating whether the pixel is land. In the sequential case, we just use *map* to apply the function, and lazily evaluate the result when *hits* gets called on it. For the parallel case we have a helper function (on line 20), a hybrid of *parMap* and *parListChunked*, which maps a given function on a list using the specified strategy for each element (rdeepseq). In addition, it takes a *chunkSize* parameter and divides the application of *map* into multiple parts, running in parallel. Note how expressive we can be about parallelism without breaching into the actual computation. *parMapChunk* is completely agnostic of the actual calculation it is going to be used on. It can be reused on similar cases

where we need chunked mapping of a function on a list.

The choice of strategy for list evaluation determines the granularity of the parallelism.

A coarse-grained variant using *rdeepseq* is used in the program and creates 4 sparks for the 4 cores in a CPU.

```
parMapChunk rdeepseq chunkSize
```

An alternative, fine-grained variant with *rpar* spawns a spark for each element, which is impractical for the example as it creates large overhead, but could be considered for smaller, computationally heavier lists.

```
parMapChunk rpar chunkSize
```

Currently the mapping process is hardcoded to be split into 4 portions for 4 cores and to use *rdeepseq*. However, it is clear that, since Strategies are created as in-program objects, we can parameterize the parallelism depending on the input problem size and number of processing elements (cores). It is possible to get this metric using *numCapabilities* from the *GHC.Conc*[5] module.

Measuring performance

A test launch of 15000 samples was done. Running the program with *+RTS -s* prints statistics of the run.

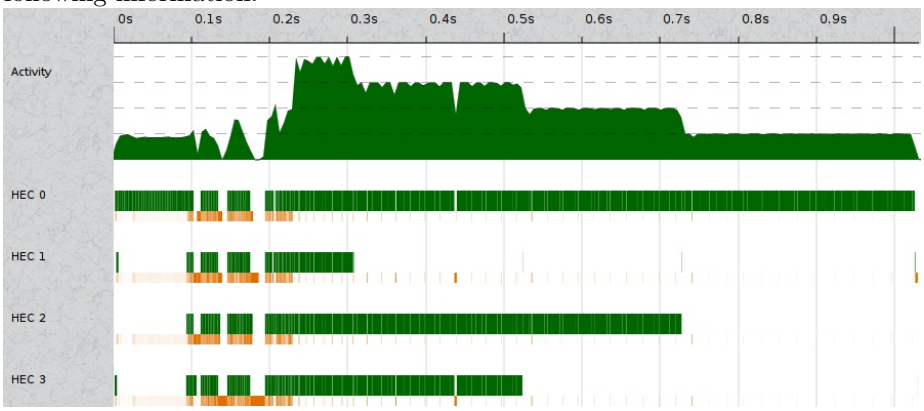
```
[cameleopard]s1139243: ./strat 15000 True +RTS -N -s
0.3675333333333333
     216,443,536 bytes allocated in the heap
      55,779,344 bytes copied during GC
      12,654,712 bytes maximum residency (7 sample(s))
       1,979,912 bytes maximum slop
              34 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       302 colls,   302 par    0.38s    0.09s     0.0003s    0.0032s
  Gen  1         7 colls,     6 par    0.15s    0.04s     0.0055s    0.0148s

  Parallel GC work balance: 21.91% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 4 (4 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    1.50s  (  0.69s elapsed)
  GC      time    0.53s  (  0.13s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.04s  (  0.83s elapsed)

  Alloc rate    144,042,959 bytes per MUT second

  Productivity  74.0% of total user, 182.5% of total elapsed

gc_alloc_block_sync: 10448
whitehole_spin: 0
gen[0].sync: 3
gen[1].sync: 22
```

We are interested in the total time and the following line:

```
SPARKS: 4 (4 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

This means that all four of the spawned sparks have been converted into parallelism. In worse cases, sparks may get fizzled, meaning their thunks were resolved by other requests before the parallel run, or garbage collected when their value was never requested at all.

To see more detailed information about the run, we can compile the program with the *-eventlog* flag and run with *+RTS -ls*, which generates an event log file. It can then be visualised with the threadscope[7] program and gives the following information.



Each *HEC* (Haskell Execution Context) is a thread, *HEC0* being the main program thread. Until 0.1s is the sequential part of the program, then it spawns three other sparks for the mapping process. Green bars show activity, while brown bars show garbage collection cycles. One can see that the sparks run for disproportionate amounts of time, even though their work is of the same size. The speedup is evident ($2.04/0.83 = 2.46\times$), but not as big an improvement as expected from 4 cores. This is most likely for the following reasons:

1. Every sample makes a memory access into a large region, which probably could not be wholly cached. This causes memory contention and slows down the program regardless of CPU efficiency. It is an issue of resources and program design, not Haskell's strategies, however.

2. There is still a sequential part in the program, which includes setting up and generating the random lists. It was measured to be roughly 0.4s, which almost matches the time of the parallel part, *coordIsLand* calls.

3. Sparking and parallel coordination introduces an overhead, which for just 4 sparks is negligible. This would leave a noticeable impact if we used the fine-grained metdod.

The largest bottleneck must have been the memory, which stays slow despite efficient CPU parallelisation. Perhaps a better example and/or several optimizations could achieve better performance gains, but the point of this experiment

was to demonstrate the flexibility and compositional power of strategies, and it shows that strategies work and achieve parallelism as intended.

Alternative methods

A well established and less extensive alternative to Strategies is the *Par* monad from *Control.Monad.Par*[6], which expresses dataflow parallelism[8]. While Strategies only work on pure computations and provide deterministic parallelism, the *Par* monad works on impure (IO()) computations as well, in which case it is nondeterministic. It would be useful for explicitly stating dependencies between parallel computations, similar to constructing the *Eval* monad manually.

Conclusion

Overall, Haskell has a clear, elegant and expressive framework for parallelism, giving the programmer a choice of how much to be involved. From applying the provided high level strategies which require almost no background knowledge to constructing new strategies from existing, which can be error prone due to not evaluating thunks fully, and finally, the *par* and *pseq* primitives can be used for complete control. The main benefits of strategies are their customisability and ease of use, particularly, application without changing at all the affected code, only adding an unobtrusive *'using'* directive.

References

[1] Marlow, Simon and Maier, Patrick and Loidl, Hans-Wolfgang and Aswad, Mustafa K. and Trinder, Phil
Seq No More: Better Strategies for Parallel Haskell.
November 2010
http://community.haskell.org/ simonmar/papers/strategies.pdf

[2] Simon Marlow
Parallel and Concurrent Programming in Haskell v1.2.
Microsoft Research Ltd., Cambridge, U.K.
May 11, 2012
http://community.haskell.org/ simonmar/par-tutorial.pdf

[3] Documentation for *Control.Parallel.Strategies* of Haskell.
The University of Glasgow
https://hackage.haskell.org/package/parallel-3.2.0.4/docs/Control-Parallel-Strategies.html

[4] JuicyPixels package
Hackage
http://hackage.haskell.org/package/JuicyPixels

[5] GHC.Conc package
Hackage
http://hackage.haskell.org/package/base-4.7.0.1/docs/GHC-Conc.html

[6] Control.Monad.Par package
Hackage
https://hackage.haskell.org/package/monad-par

[7] The ThreadScope program description
Haskell Wiki
https://www.haskell.org/haskellwiki/ThreadScope

[8] Simon Marlow/Facebook
Parallel Haskell with the Par Monad
http://www.cse.chalmers.se/edu/course/pfp/lectures/lecture2/Marlow14.pdf

[9] Maps encyclopedia from facts.co
Map nr. 2
http://facts.co/scotland/scotlandmapof/scotlandmap.php

All web links were verified to be working on 07.11.2014