Parallelism in Haskell

Parallel computing

Meelis Utt

2020-11-11

Introduction

For this report I chose Haskell programming language, as I am currently very interested in it. Haskell is stronly general-purpose, strongly typed, lazily evaluated purely functional programming language (1). Haskell separates pure functions and functions with side efects. Pure function is a function, that gives same output for same input every time, meaning it is deterministic (4). Side efect can be printing to standard output, sending data over network, generating random number etc. It started in the academia, meaning scientist were the ones who mostly developed it in the beginning. In Haskell there is two types of parallelism: pure parallelism and concurrency (2). I also found packages (haskell-mpi,mpi-hs), that use MPI for parallelization. I found some mentions of OpenMP in Haskell, but I did not research this currently. I planned on using MPI solution at first, but I had a dependecy issue, that I was unable to not resolve. Since the example of Monte-Carlo solution I made for this report is more aligned with pure parallelism, then I will be focusing more on this. Although, reading the materials gave me an impression, that concurrency is really useful in networking (3). I would like to share link to a video about concurrency in haskell, that I found really useful and interesting (5).
I used Haskell package called parallel, which is part of the pure parallelism I mentioned before. Pure parallelism has the advantages of

- Guaranteed deterministic (same result every time);
- no race conditions or deadlocks (2)

Since Haskell is lazy language and has immutable variables, then it is possible to take a normal Haskell function, apply a few simple transformations to it, and have it evaluated in parallel (3). This video (6) helped me understand this subject better and gave some examples of good and bad parallelism. When making my parallel implementation of this Monte-Carlo example, I had a lot of trouble of getting the implementation reasonable in a way that gave me speedup. In Haskell package *parallel* the parallelism is handled by the runtime system (RTS) and things calles *Spark*'s are used. Spark is something that takes some unevaluated data and evaluates it in parallel. When spark is created, it is put into the *spark pool*, which is Haskell Execution Contect (HEC). One HEC is roughly equivalent to one core on ones machine. For more indepth overview of sparks, I once again recommend this (6) video. Before moving on to the examples, I want to mention that there is a greate tool (7) for spark event analysis, that I unfortunatly did not use during this example. Also, I want to mention that since the package is currently marked (at the time of writing) experimental, then there exists possibility that some functions may change. However, since the package is fairly popular, then there is small probability for a major change.

Example

First let's look at the setup of the example. In this example, we want to find the mean value of the function

$$f(x) = x^2 + x^4 + \sin(x) + \cos(x) + x^2 5.$$

This function was chosen, because it is fairly simple to find the mean analytically, but it still gives some computational complexity when using the Monte-Carlo method. The mean can be calculate analytically using the formula

$$E(f(x)) = \int_a^b f(x)dx.$$

In my example, I used uniform distribution $X \sim U(0,1)$ to generate the random values in the Monte-Carlo method. So the analytically we get

$$E(f(x)) = \int_0^1 f(x)dx = \int_0^1 x^2 + x^4 + \sin(x) + \cos(x) + x^2 5dx = \frac{613}{390} + \sin(1) - \cos(1) \approx 1.8729635507346285.$$

I implemented the function, analytical solution and generator of uniform distribution values separately from the MC examples and imported compiled code. I compile both serial and parallel code with command

```
stack ghc -- -threaded -rtsopts -eventlog -main-is MC<type> MC<type>.hs
```

and ran with command

```
./MC<type> <n> +RTS -N
or
./MC<type> <n> +RTS -N -s
```

In case of serial code, the flags *+RTS* and textit{-N} do nothing, but for the sake of comparability I added them. In case of parallel program, the flag *-N* specifies how many cores are used. If there is no number specified in the flag (eg -N2), then the maximum number of cores are used. The *-s* option gives more info about the running of the program and *-ls* would create a file, that could be used in the spark analysis tool I mentioned. For example, the flag *-s* gives us output

```
./MCserial 100 +RTS -N -s
```

```
## result,error,analytical
## 1.859461716002568,1.350183473206057e-2,1.8729635507346285
##        2,041,664 bytes allocated in the heap
##          194,304 bytes copied during GC
##           89,824 bytes maximum residency (1 sample(s))
##           57,632 bytes maximum slop
##                0 MB total memory in use (0 MB lost due to fragmentation)
##
##                                     Tot time (elapsed)  Avg pause  Max pause
##   Gen  0         1 colls,     1 par    0.000s   0.000s     0.0003s    0.0003s
##   Gen  1         1 colls,     0 par    0.000s   0.000s     0.0002s    0.0002s
##
##   Parallel GC work balance: 15.55% (serial 0%, perfect 100%)
##
##   TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
##
##   SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
##
##   INIT    time    0.000s  (  0.001s elapsed)
##   MUT     time    0.001s  (  0.004s elapsed)
##   GC      time    0.000s  (  0.000s elapsed)
##   EXIT    time    0.000s  (  0.005s elapsed)
##   Total   time    0.001s  (  0.011s elapsed)
##
##   Alloc rate    1,396,487,004 bytes per MUT second
##
##   Productivity 100.0% of total user, 41.9% of total elapsed
```

```
./MCparallel 100 +RTS -N -s
```

```
## result,error,analytical
## 1.8619773801555939,1.8729635507346285,1.0986170579034615e-2
##       34,357,672 bytes allocated in the heap
##        1,302,744 bytes copied during GC
##          227,424 bytes maximum residency (2 sample(s))
##           61,712 bytes maximum slop
##                0 MB total memory in use (0 MB lost due to fragmentation)
##
##                                     Tot time (elapsed)  Avg pause  Max pause
##   Gen  0        12 colls,    12 par    0.026s   0.002s     0.0002s    0.0004s
##   Gen  1         2 colls,     1 par    0.002s   0.001s     0.0003s    0.0005s
##
##   Parallel GC work balance: 29.55% (serial 0%, perfect 100%)
##
##   TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
##
##   SPARKS: 50 (49 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)
##
##   INIT    time    0.000s  (  0.001s elapsed)
##   MUT     time    0.084s  (  0.032s elapsed)
##   GC      time    0.028s  (  0.003s elapsed)
##   EXIT    time    0.001s  (  0.006s elapsed)
##   Total   time    0.113s  (  0.041s elapsed)
##
##   Alloc rate    406,907,859 bytes per MUT second
##
##   Productivity  74.5% of total user, 77.7% of total elapsed
```

As we can see from the example, the serial code was faster than the parallel code in case of 100 iterations ($0.011sec$ vs $0.041sec$ elapsed). However, in Monte-Carlo method it is usual to run the code with bigger n. Let's try running the code with $n = 10^3, 10^4, \ldots, 10^7$. We get

Code

Function code:

```
module Function where

-- Define a function, that has some complexity,
-- so that parallel computation might give better results,
-- but is still fairly simple to analytically calculate the exact answer.
f :: Double -> Double
f x = x^2 + x^4 + (sin x) + (cos x) + (x^25)

-- We assume that random variables are taken from uniform distribution U(0,1).
-- This means we can calculate the mean (EX) as
-- integral from 0 to 1 of the function f(x).
analytical :: Double
analytical = 613/390 + (sin 1) - (cos 1)
```

Generator code:

```
module GenUnif where

import Control.Monad (replicateM)
import Control.Monad.Random (uniform)

size :: Double
size = 1000000

lower :: Double
lower = 0
upper :: Double
upper = 1
step :: Double
step = 0.001

valList :: [Double]
valList = [lower,lower+step..upper]

unif :: Int -> IO [Double]
unif n = replicateM n $ uniform vals
 where
  vals = valList
```

Serial implementation:

```
module MCserial where

import System.Environment (getArgs)
import GenUnif
import Function

mc :: (Integer,Double) -> [Double] -> Double
mc (n,summed) []     = summed/(fromIntegral n)
mc (n,summed) (x:xs) = mc (n+1,summed+(f x)) xs

main = do
 -- let n = 10000
```

```
 -- read number of mc iterations n.
 [nstr] <- getArgs
 let n = read nstr :: Int
 -- generate list of n random values from uniform distribution U(0,1).
 xs <- unif n
 -- calculate mc result
 let result = mc (0,0) xs
 -- calculate error
 let error = abs $ result - analytical
 -- check if error >0.05. If not, then print the result.
 if (error > 0.05) then do
  putStrLn "Incorrect answer, error > 0.05!"
 else do
  putStrLn "result,error,analytical"
  putStrLn $ (show result) ++ "," ++ (show error)++ "," ++ (show analytical)
```

Parallel implementation:

```
module MCparallel where

import System.Environment (getArgs)
import Control.Parallel.Strategies
import Control.Monad
import GenUnif
import Function

mc :: (Integer,Double) -> [Double] -> Double
mc (n,summed) []     = summed/(fromIntegral n)
mc (n,summed) (x:xs) = mc (n+1,summed+(f x)) xs

main = do
 -- read number of mc iterations n and the number
 -- that we use to divide n into chunks
 -- In my testing I found that best speedup is achieved,
 -- if n/p is in the interval [25,50].
 -- However, using n/p=100 gives more stable answer,
 -- meaning error more than 0.05 is less likely.
 -- Also, correct answer is calculated in case of with smaller n.
 -- [nstr,pstr] <- getArgs
 [nstr] <- getArgs
 let n = read nstr :: Int
 -- let p = read pstr :: Int
 let p = (div) n 50
 let nchunk = (div) n p
 -- make double nchuck value.
 -- Since we find means of sublists,
 -- then getting the correct final result can be calculated as
 -- sum(intermediate means)/nchuck
 let nchunkd = fromIntegral nchunk
 -- generate list of list with total of n random values
 -- from uniform distribution U(0,1)
 xs <- replicateM nchunk (unif nchunk)
 -- calculate intermediate means.
 let resultchunk = parMap rseq (\ys -> mc (0,0) ys) xs
 -- calculate final result.
```

```
let result = sum(resultchunk)/nchunkd
-- calculate error
let error = abs $ result - analytical
print (result,analytical,error)
-- check if error >0.05. If not, then print the result.
if (error > 0.05) then do
 putStrLn "Incorrect answer, error > 0.05!"
else do
 putStrLn "result,error,analytical"
 putStrLn $ (show result) ++ "," ++ (show analytical) ++ "," ++ (show error)
```

Used literature

1. http://book.realworldhaskell.org/, 2020-11-11
2. https://wiki.haskell.org/Parallelism, 2020-11-11
3. http://book.realworldhaskell.org/read/concurrent-and-multicore-programming.html, 2020-11-11
4. https://wiki.haskell.org/Pure, 2020-11-11
5. https://www.youtube.com/watch?v=cuHD2qTXxL4, 2020-11-11
6. https://www.youtube.com/watch?v=R47959rD2yw, 2020-11-11
7. https://wiki.haskell.org/ThreadScope_Tour/Spark, 2020-11-11