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 currently research this. 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 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 the package *parallel* the parallelism is handled by the runtime system (RTS). Parallel work is done by *spark*'s. Spark is something that takes some unevaluated data and evaluates it in parallel. When a spark is created, it is put into the *spark pool*, which is the 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

$$f(x) = x^2 + x^4 + \sin(x) + \cos(x) + x^{25}.$$

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, generator of uniform distribution values and timing functinos separately from the MC examples and imported the compiled code to serial and parallel examples. At first I parallelised the calculation of mean. Later I also tried to parallelize the generation of random numbers. However, I did not succeed in getting very good parallelization. I compile both serial and parallel code with command

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

and the executable can be run with

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

In case of serial code, the flags *+RTS* and *-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
```

```
## Incorrect answer, error > 0.05!
##        1,986,568 bytes allocated in the heap
##          231,776 bytes copied during GC
##           85,968 bytes maximum residency (1 sample(s))
##           61,488 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    0.010s   0.006s     0.0062s    0.0062s
##   Gen  1         1 colls,     0 par    0.000s   0.000s     0.0003s    0.0003s
##
##   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    0.002s  (  0.011s elapsed)
##   MUT     time    0.006s  (  0.018s elapsed)
##   GC      time    0.011s  (  0.007s elapsed)
##   EXIT    time    0.001s  (  0.008s elapsed)
##   Total   time    0.020s  (  0.042s elapsed)
##
##   Alloc rate    307,613,502 bytes per MUT second
##
##   Productivity  33.1% of total user, 41.8% of total elapsed
```

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

```
## result,error,analytical
## 1.8695138078936424,1.8729635507346285,3.449742840986092e-3
##        2,148,584 bytes allocated in the heap
##          187,312 bytes copied during GC
##           89,568 bytes maximum residency (1 sample(s))
##           61,984 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   0.001s     0.0005s    0.0005s
##   Gen  1         1 colls,     0 par    0.000s   0.000s     0.0005s    0.0005s
##
##   Parallel GC work balance: 27.98% (serial 0%, perfect 100%)
##
##   TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
##
##   SPARKS: 1101 (1078 converted, 0 overflowed, 0 dud, 22 GC'd, 1 fizzled)
##
##   INIT    time    0.000s  (  0.005s elapsed)
##   MUT     time    0.009s  (  0.009s elapsed)
##   GC      time    0.001s  (  0.001s elapsed)
##   EXIT    time    0.001s  (  0.006s elapsed)
##   Total   time    0.011s  (  0.021s elapsed)
##
##   Alloc rate    229,060,127 bytes per MUT second
##
##   Productivity  85.8% of total user, 42.0% of total elapsed
```

As we can see from this example, the codes ran roughly at the same speed (although the error on parallel program might be bit more than 0.05 in some cases). However, in Monte-Carlo method it is usual to run the code with bigger n. Let's try running the code with $n = 10^3, 5 \cdot 10^3, 10^4, 5 \cdot 10^4, \ldots, 10^7$. We get

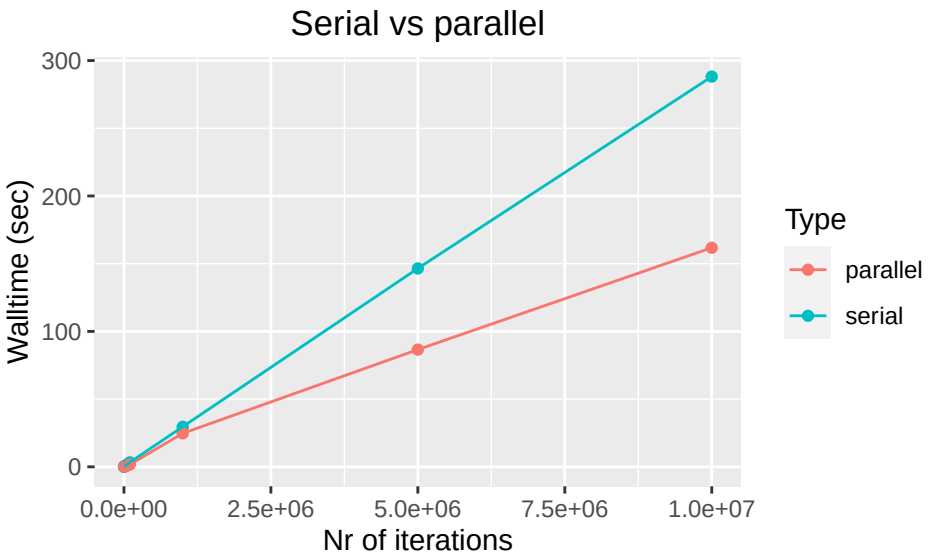
```
rm -f results.csv
for n in 1000 5000 10000 50000 100000 1000000 5000000 10000000
do
  ./MCserial $n +RTS -N >> results.csv
  ./MCparallel $n +RTS -N >> results.csv
done
```

```
data <- read.csv("results.csv",header=F,stringsAsFactors=F) %>%
  setNames(c("n","result","analytical","error","type","time"))
data
```

```
##           n   result analytical        error     type        time
## 1      1000 1.861324   1.872964 0.0116397276   serial   0.0648096
## 2      1000 1.909696   1.872964 0.0367325282 parallel   0.0466885
## 3      5000 1.875816   1.872964 0.0028522372   serial   0.2063854
## 4      5000 1.878995   1.872964 0.0060317900 parallel   0.1581338
## 5     10000 1.871065   1.872964 0.0018986704   serial   0.3979648
## 6     10000 1.883455   1.872964 0.0104914421 parallel   0.3333372
## 7     50000 1.876721   1.872964 0.0037575355   serial   1.6177962
## 8     50000 1.874714   1.872964 0.0017504185 parallel   0.9044851
## 9    100000 1.873273   1.872964 0.0003097205   serial   3.2957750
## 10   100000 1.874158   1.872964 0.0011939641 parallel   1.6673656
```

```
## 11  1000000 1.873471   1.872964 0.0005070108   serial  29.6040841
## 12  1000000 1.873335   1.872964 0.0003714020 parallel  24.7128204
## 13  5000000 1.873509   1.872964 0.0005457538   serial 146.4985924
## 14  5000000 1.873381   1.872964 0.0004173063 parallel  86.5756337
## 15 10000000 1.873955   1.872964 0.0009909975   serial 288.1460522
## 16 10000000 1.873456   1.872964 0.0004919569 parallel 161.7782657
```

```
ggplot(data=data, aes(x=n,y=time,color=type,group=type)) +
  geom_point() +
  geom_line() +
  labs(
       title="Serial vs parallel",
       x="Nr of iterations",
       y="Walltime (sec)"
       ) +
  theme(
        plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)
       ) +
  guides(color=guide_legend(title="Type"))
```



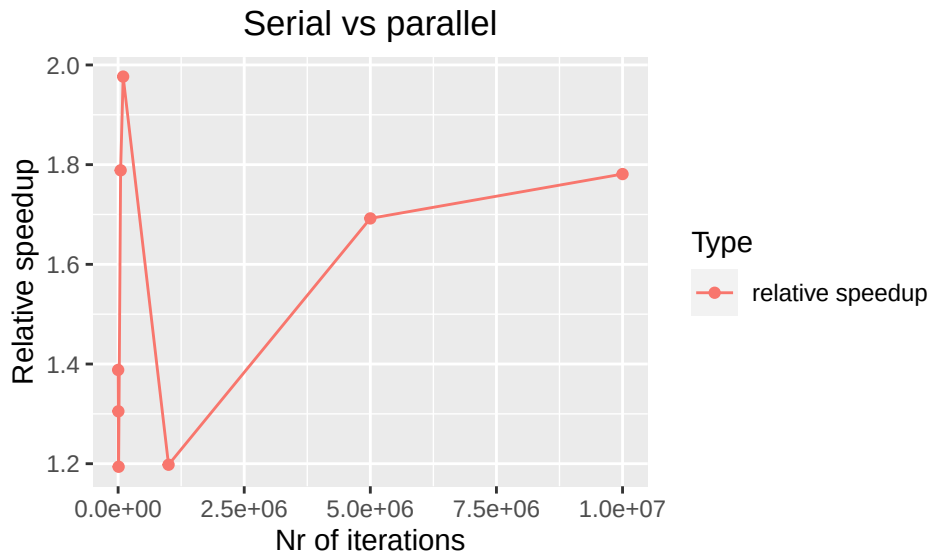
The parallel solution is faster in case of all the n values. For smaller n values it is difficult to see the difference on the plot. Let's look at the relative speedup.

```
par <- data[data$type=="parallel",]
ser <- data[data$type=="serial",]
relative <- dplyr::inner_join(par,ser,by="n") %>%
  mutate(
    type = "relative speedup",
    relative.speedup = time.y/time.x
  ) %>%
  select(n,type,relative.speedup)
relative
```

```
##         n             type relative.speedup
## 1    1000 relative speedup         1.388128
## 2    5000 relative speedup         1.305132
```

```
## 3    10000 relative speedup         1.193880
## 4    50000 relative speedup         1.788638
## 5   100000 relative speedup         1.976636
## 6  1000000 relative speedup         1.197924
## 7  5000000 relative speedup         1.692146
## 8 10000000 relative speedup         1.781117
```

```
ggplot(data=relative, aes(x=n,y=relative.speedup,color=type,group=type)) +
  geom_point() +
  geom_line() +
  labs(
       title="Serial vs parallel",
       x="Nr of iterations",
       y="Relative speedup"
       ) +
  theme(
        plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)
       ) +
  guides(color=guide_legend(title="Type"))
```



As we can see, the speedup converges arount 1.8. The best speedup we gained was 1.9766361.
I did not achieve very good speedup with this example. I think it is possible to get better speedups, if generation of random numbers is parallelized better. Furthermore, I think that generation of random numbers is done in an inefficient why. Unfortunately, I was unable to improve the generation. To put into perspective, I put together a simple serial and parallel MC method implementation in R. Since R is dynamic language, it should be slower. Let's compare the relative speedups of the quick serial and parallel R implementation against this Haskell example. We get

```
f <- function(x){
  return(x**2 + x**4+ sin(x) + cos(x) +x**25)
}
analytical <- 613/390 + sin(1) - cos(1)
powers <- 3:7
iterations <- rep(c(1,5),each=length(powers))*10**(powers)
iterations <- iterations[-length(iterations)]
```

```
# serial
null <- lapply(iterations,function(n){
    start <- Sys.time()
    i <- runif(n,0,1)
    EX <- mean(f(i))
    end <- Sys.time()
    time <- difftime(end,start)
    error <- EX - analytical
    write.table(file = "results.csv",
                x = paste(n,EX,analytical,error,"R serial",time,sep = ","),
                append = TRUE,col.names=F,row.names=F,quote=F)
  }
)

library(parallel)
#parallel
null <- sapply(iterations,function(n){
  start <- Sys.time()
  # Calculate the number of cores
  no_cores <- detectCores()
  start <- Sys.time()

  # Initiate cluster
  cl <- makeCluster(no_cores)

  intermean <- parSapply(cl, rep(n/no_cores,no_cores),function(n,f){
    i <- runif(n,0,1)
    EX <- mean(f(i))

  },f
  )
  stopCluster(cl)
  EX <- mean(intermean)
  end <- Sys.time()
  time <- difftime(end,start)
  error <- EX - analytical
    write.table(file = "results.csv",
                x = paste(n,EX,analytical,error,"R parallel",time,sep = ","),
                append = TRUE,col.names=F,row.names=F,quote=F)
})
```

```
data <- read.csv("results.csv",header=F,stringsAsFactors=F) %>%
  setNames(c("n","result","analytical","error","type","time"))
data %>% arrange(n,type)
```

```
##        n   result analytical         error       type         time
## 1  1e+03 1.909696   1.872964  3.673253e-02   parallel 4.668850e-02
## 2  1e+03 1.902297   1.872964  2.933312e-02 R parallel 4.459112e-01
## 3  1e+03 1.895744   1.872964  2.278020e-02   R serial 1.621008e-03
## 4  1e+03 1.861324   1.872964  1.163973e-02     serial 6.480960e-02
## 5  5e+03 1.878995   1.872964  6.031790e-03   parallel 1.581338e-01
## 6  5e+03 1.888996   1.872964  1.603202e-02 R parallel 4.352014e-01
## 7  5e+03 1.870885   1.872964 -2.078231e-03   R serial 6.971359e-04
## 8  5e+03 1.875816   1.872964  2.852237e-03     serial 2.063854e-01
```

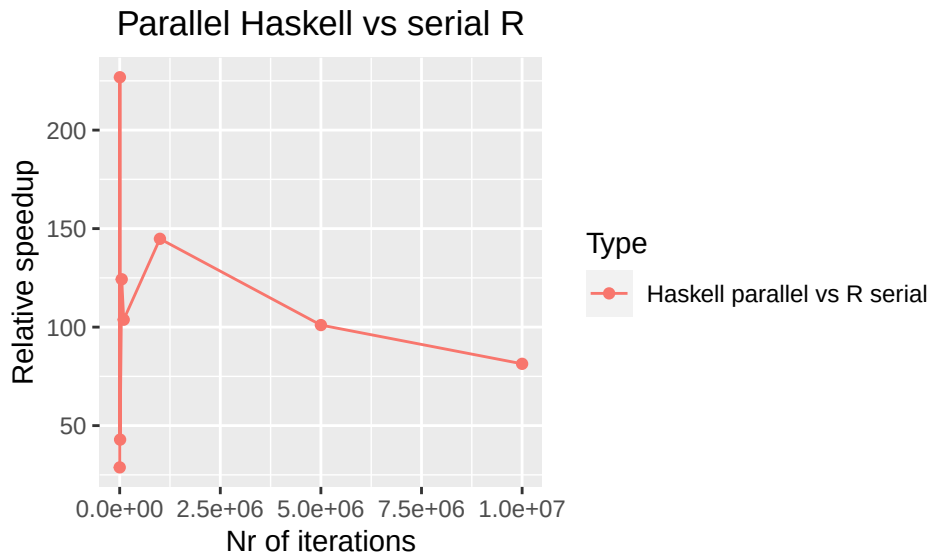
```
## 9  1e+04 1.883455   1.872964  1.049144e-02   parallel 3.333372e-01
## 10 1e+04 1.871583   1.872964 -1.380405e-03 R parallel 4.541450e-01
## 11 1e+04 1.878403   1.872964  5.439559e-03   R serial 7.770061e-03
## 12 1e+04 1.871065   1.872964  1.898670e-03     serial 3.979648e-01
## 13 5e+04 1.874714   1.872964  1.750418e-03   parallel 9.044851e-01
## 14 5e+04 1.875648   1.872964  2.684686e-03 R parallel 4.526839e-01
## 15 5e+04 1.873005   1.872964  4.158096e-05   R serial 7.276773e-03
## 16 5e+04 1.876721   1.872964  3.757535e-03     serial 1.617796e+00
## 17 1e+05 1.874158   1.872964  1.193964e-03   parallel 1.667366e+00
## 18 1e+05 1.876384   1.872964  3.420533e-03 R parallel 4.510272e-01
## 19 1e+05 1.869342   1.872964 -3.621956e-03   R serial 1.606655e-02
## 20 1e+05 1.873273   1.872964  3.097205e-04     serial 3.295775e+00
## 21 5e+05 1.873966   1.872964  1.002037e-03 R parallel 4.811442e-01
## 22 5e+05 1.873639   1.872964  6.750334e-04   R serial 8.002853e-02
## 23 1e+06 1.873335   1.872964  3.714020e-04   parallel 2.471282e+01
## 24 1e+06 1.873638   1.872964  6.742128e-04 R parallel 5.156527e-01
## 25 1e+06 1.872685   1.872964 -2.789040e-04   R serial 1.706431e-01
## 26 1e+06 1.873471   1.872964  5.070108e-04     serial 2.960408e+01
## 27 5e+06 1.873381   1.872964  4.173063e-04   parallel 8.657563e+01
## 28 5e+06 1.873140   1.872964  1.765154e-04 R parallel 8.258994e-01
## 29 5e+06 1.872893   1.872964 -7.085187e-05   R serial 8.568411e-01
## 30 5e+06 1.873509   1.872964  5.457538e-04     serial 1.464986e+02
## 31 1e+07 1.873456   1.872964  4.919569e-04   parallel 1.617783e+02
## 32 1e+07 1.873132   1.872964  1.686372e-04 R parallel 1.293747e+00
## 33 1e+07 1.872924   1.872964 -3.998205e-05   R serial 1.988024e+00
## 34 1e+07 1.873955   1.872964  9.909975e-04     serial 2.881461e+02
```

```
par <- data[data$type=="parallel",]
Rser <- data[data$type=="R serial",]
Rpar <- data[data$type=="R parallel",]

parRser <- dplyr::inner_join(Rser,par,by="n") %>%
  mutate(
    type = "Haskell parallel vs R serial",
    relative.speedup = time.y/time.x
  ) %>%
  select(n,type,relative.speedup)

parRpar <- dplyr::inner_join(Rpar,par,by="n") %>%
  mutate(
    type = "Haskell parallel vs R parallel",
    relative.speedup = time.y/time.x
  ) %>%
  select(n,type,relative.speedup)

RparRser <- dplyr::inner_join(Rpar,Rser,by="n") %>%
  mutate(
    type = "serial vs parallel R",
    relative.speedup = time.y/time.x
  ) %>%
  select(n,type,relative.speedup)

relative <- dplyr::union_all(parRser,parRpar) %>%
  dplyr::union_all(RparRser)
```

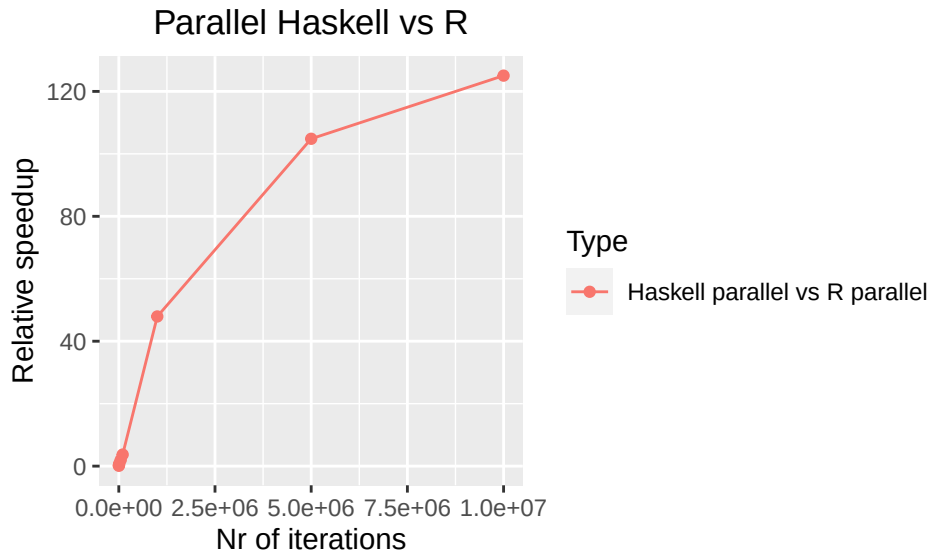
```
relative
```

```
##        n                          type relative.speedup
## 1  1e+03   Haskell parallel vs R serial     2.880214e+01
## 2  1e+04   Haskell parallel vs R serial     4.290021e+01
## 3  1e+05   Haskell parallel vs R serial     1.037787e+02
## 4  1e+06   Haskell parallel vs R serial     1.448217e+02
## 5  1e+07   Haskell parallel vs R serial     8.137641e+01
## 6  5e+03   Haskell parallel vs R serial     2.268335e+02
## 7  5e+04   Haskell parallel vs R serial     1.242976e+02
## 8  5e+06   Haskell parallel vs R serial     1.010405e+02
## 9  1e+03 Haskell parallel vs R parallel     1.047036e-01
## 10 1e+04 Haskell parallel vs R parallel     7.339886e-01
## 11 1e+05 Haskell parallel vs R parallel     3.696819e+00
## 12 1e+06 Haskell parallel vs R parallel     4.792532e+01
## 13 1e+07 Haskell parallel vs R parallel     1.250463e+02
## 14 5e+03 Haskell parallel vs R parallel     3.633577e-01
## 15 5e+04 Haskell parallel vs R parallel     1.998050e+00
## 16 5e+06 Haskell parallel vs R parallel     1.048259e+02
## 17 1e+03           serial vs parallel R     3.635271e-03
## 18 1e+04           serial vs parallel R     1.710921e-02
## 19 1e+05           serial vs parallel R     3.562214e-02
## 20 1e+06           serial vs parallel R     3.309264e-01
## 21 1e+07           serial vs parallel R     1.536641e+00
## 22 5e+03           serial vs parallel R     1.601870e-03
## 23 5e+04           serial vs parallel R     1.607473e-02
## 24 5e+05           serial vs parallel R     1.663296e-01
## 25 5e+06           serial vs parallel R     1.037464e+00
```

```
ggplot(data=parRser, aes(x=n,y=relative.speedup,color=type,group=type)) +
  geom_point() +
  geom_line() +
  labs(
       title="Parallel Haskell vs serial R",
       x="Nr of iterations",
       y="Relative speedup"
       ) +
  theme(
        plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)
       ) +
  guides(color=guide_legend(title="Type"))
```

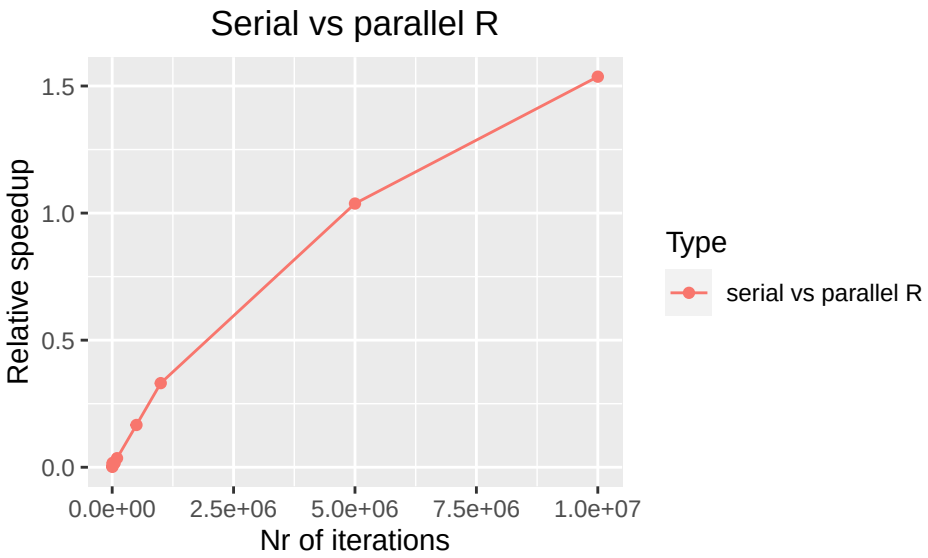



```
ggplot(data=parRpar, aes(x=n,y=relative.speedup,color=type,group=type)) +
  geom_point() +
  geom_line() +
  labs(
       title="Parallel Haskell vs R",
       x="Nr of iterations",
       y="Relative speedup"
       ) +
  theme(
        plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5)
       ) +
  guides(color=guide_legend(title="Type"))
```




```
ggplot(data=RparRser, aes(x=n,y=relative.speedup,color=type,group=type)) +
  geom_point() +
  geom_line() +
```

```
labs(
     title="Serial vs parallel R",
     x="Nr of iterations",
     y="Relative speedup"
     ) +
theme(
      plot.title = element_text(hjust = 0.5),
      plot.subtitle = element_text(hjust = 0.5)
     ) +
guides(color=guide_legend(title="Type"))
```



As we can see, serial R example has better speedup, when n was smaller. However, when n got bigger, the parallel implementation was faster.

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