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 and it gave some speedup. Later I parallelised also the generation of random numbers. This gave me better speedup, but gives more unstable answers, when n is small (this might be seen from the numberical example as well). 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!
##       2,036,368 bytes allocated in the heap
##         229,552 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    0.002s   0.000s     0.0004s    0.0004s
##   Gen  1         1 colls,     0 par    0.000s   0.000s     0.0002s    0.0002s
##
##   Parallel GC work balance: 29.68% (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    0.001s  (  0.001s elapsed)
##   MUT     time    0.004s  (  0.004s elapsed)
##   GC      time    0.002s  (  0.001s elapsed)
##   EXIT    time    0.000s  (  0.006s elapsed)
##   Total   time    0.007s  (  0.011s elapsed)
##
##   Alloc rate    525,243,229 bytes per MUT second
```

```
##
##   Productivity  53.1% of total user, 33.3% of total elapsed
```

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

```
## Incorrect answer, error > 0.05! Error: 0.13192297077121395
##        1,172,480 bytes allocated in the heap
##               56 bytes copied during GC
##           90,080 bytes maximum residency (1 sample(s))
##           53,280 bytes maximum slop
##                0 MB total memory in use (0 MB lost due to fragmentation)
##
##                                     Tot time (elapsed)  Avg pause  Max pause
##   Gen  0         0 colls,     0 par    0.000s   0.000s     0.0000s    0.0000s
##   Gen  1         1 colls,     0 par    0.000s   0.000s     0.0002s    0.0002s
##
##   TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
##
##   SPARKS: 50 (50 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
##
##   INIT    time    0.000s  (  0.001s elapsed)
##   MUT     time    0.000s  (  0.001s elapsed)
##   GC      time    0.000s  (  0.000s elapsed)
##   EXIT    time    0.000s  (  0.009s elapsed)
##   Total   time    0.000s  (  0.011s elapsed)
##
##   Alloc rate    0 bytes per MUT second
##
##   Productivity 100.0% of total user, 9.0% 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 $n = 10^3, 5 \cdot 10^3, 10^4, 5 \cdot 10^4, \ldots, 50^6$. We get

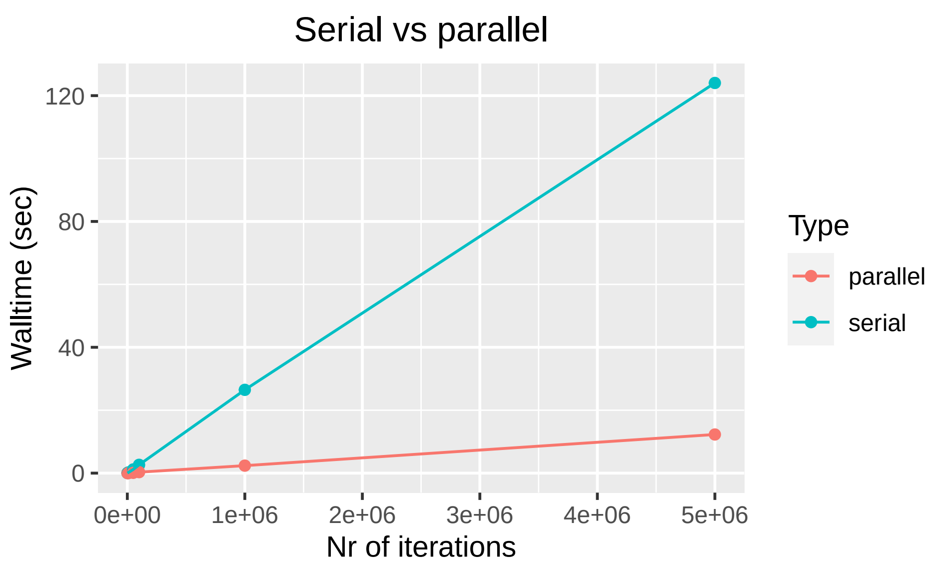
```
rm -f results.csv
for n in 1000 5000 10000 50000 100000 1000000 5000000
do
  ./MCserial $n +RTS -N >> results.csv
  ./MCparallel $n +RTS -N >> results.csv
done
```

```
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 1.855648   1.872964 0.0173152957   serial 2.364616e-02
## 2     1000 2.042525   1.872964 0.1695611107 parallel 3.172758e-03
## 3     5000 1.871959   1.872964 0.0010042679   serial 1.178287e-01
## 4     5000 2.024125   1.872964 0.1511611473 parallel 1.150365e-02
## 5    10000 1.876927   1.872964 0.0039634102   serial 2.306937e-01
## 6    10000 1.881943   1.872964 0.0089791540 parallel 1.948778e-02
## 7    50000 1.874550   1.872964 0.0015860195   serial 1.168226e+00
## 8    50000 1.861271   1.872964 0.0116925352 parallel 9.227584e-02
## 9   100000 1.875662   1.872964 0.0026983386   serial 2.638401e+00
## 10  100000 1.876644   1.872964 0.0036800923 parallel 3.139828e-01
## 11 1000000 1.874043   1.872964 0.0010790570   serial 2.647813e+01
```

```
## 12 1000000 1.876485   1.872964 0.0035213721 parallel 2.383466e+00
## 13 5000000 1.873464   1.872964 0.0005006455   serial 1.240260e+02
## 14 5000000 1.872832   1.872964 0.0001315278 parallel 1.228122e+01
```

```
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"))
```



As we can see, the parallel solution is faster in case of all the n values. We see bigger difference in the walltime starting with $n = 50000$ 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         7.452873
## 2    5000 relative speedup        10.242715
## 3   10000 relative speedup        11.837866
## 4   50000 relative speedup        12.660147
## 5  100000 relative speedup         8.403011
```

```
## 6 1000000 relative speedup        11.109087
## 7 5000000 relative speedup        10.098837
```

As we can see, we gain better speedups when n increases.
Now let's try running the serial and parallel code in the course VM. We get

```r
# data <- read.csv("results_VM.csv",header=F,stringsAsFactors=F) %>%
#   setNames(c("n","result","analytical","error","type","time"))
# data
# ggplot(data=data, aes(x=n,y=time,color=type,group=type)) +
#   geom_point() +
#   geom_line() +
#   labs(
#        title="Serial vs parallel in VM",
#        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="Walltime"))
```

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