R programming language: parallel computations

Maxim Litvak

2016-06-10

Outline

A task to produce result Y out of input X may have steps that could be performed in parallel

Example: you need to drink a 1L of beer

Ask a friend to help

The taks is, thus, parallelized among two throats

you can split the tasks in parallel steps, but who does the job?

Nowadays, each PC usually has 2-8 CPU cores.

Each core can process a task!

You can split the tasks across different computers

You can rent them at Microsoft, Amazon etc

Map step - assign to each work a task to perform

Reduce step - collect results from different workers and produce the overall result

Random numbers used for simulations are "pseudo-random", i.e. a sequence of number that looks random

The starting point is called a "seed". Same seed - same sequence

Any simulation system must be reproducible

assume we need to calculate 10 million of random numbers

if we can split it among 4 CPU cores than each of them needs to generate only 2.5 millions

we can assign to each core a different seed, generate the numbers and unite the numbers into one

Loss distribution - non-parallel

```
loss.dist <- function(seed, N)
{
  set.seed(seed)
  return(runif(N))
}

N <- 4
seed <- 1
print(loss.dist(seed, N))
```

```
[1] 0.2655087 0.3721239 0.5728534 0.9082078
```

Explore your ressources

```
library(parallel)
print(detectCores())
```

```
[1] 8
```

Loss distribution - parallel

```
N <- 4
seed <- 1
print(loss.dist(seed, N))
```

A lot of implementation details are hidden in R (e.g. compared to C#)

General schema

Create a cluster (makeCluster)

Put in the scope of cluster objects (functions, variables etc) which are needed there (clusterExport)

Send a task (function together with input) to the cluster (parLapply) - similar to apply functions

Take a trivial task as an example - calculate sum of the sample

divide N numbers sample on M workers

calculate the sums for each workers (in parallel)

collect the results and calculate their sum

```
nc <- 2 # number of cores
cl <- makeCluster(no_cores)
sq <- 1:8
L <- length(sq)
clusterExport(cl, list("sq", "nc", "L"))
# check first that the split is correct
parLapply(cl, 1:nc
  , function(x) sq[(1+(x-1)*L/nc):(x*L/nc)]
)
# [[1]]
# [1] 1 2 3 4
# [[2]]
# [1] 5 6 7 8


res <- parLapply(cl, 1:nc
, function(x) sum(sq[(1+(x-1)*L/nc):(x*L/nc)])
)
print(res)
```

```
# [[1]]
# [1] 10
# [[2]]
# [1] 26
print(sum(unlist(res))) # collect results
# [1] 36
```

Farenheit to Celcius

Take this as input (correct parLapply call is to be implemented)

```
library(parallel)
c <- function(t) t*5/9-32
nc <- 4
temps <- seq(10, 40, 10)
cl <- makeCluster(nc)
clusterExport(cl, list("temps"))
```

Example attempt

parLapply(cl, 1:nc, function(x) c(temps[x]))

However, temperatures are still in Farenheit, what is wrong here?

Try to correct it

```
library(parallel)
C <- function(t) t*5/9-32
nc <- 4
temps <- seq(10, 40, 10)
cl <- makeCluster(nc)
clusterExport(cl, list("temps", "C"))
parLapply(cl, 1:nc, function(x) C(temps[x]))

[[1]]
[1] -26.44444

[[2]]
[1] -20.88889

[[3]]
[1] -15.33333

[[4]]
[1] -9.777778
```

standard object c was overwritten and wasn't put in the scope of cluster

Even worse it didn't throw an error since the cluster used the default object

Recommended to review the topic on scope in

https://github.com/maxlit/workshops/blob/master/R/r-advanced-overview/r-workshop-general-overview.pdf

Be careful with the scope of the cluster

distribute carefully among work loaders

consider a simple loss generation

```
loss.dist <- function(seed, N)
{
  set.seed(seed)
  return(runif(N))
}
print(loss.dist(1,4))
# [1] 0.2655087 0.3721239 0.5728534 0.9082078

[1] 0.2655087 0.3721239 0.5728534 0.9082078
```

How to parallelize it?

Possible solution

```
parallel.loss.dist <- function(seed, N)
{
  no_cores <- 2
  cl <- makeCluster(no_cores)
  clusterExport(cl, list("loss.dist"))
  temp.res <- parLapply(
cl
, seed:(seed + 1)
, function(x) loss.dist(x, N)
)
  stopCluster(cl)
  return(unlist(temp.res))
}
print(parallel.loss.dist(1,4))
# [1] 0.2655087 0.3721239 0.1848823 0.7023740
```

Why the first 2 numbers coincide with non-parallel version and the rest not?

Where is the "Map" step and where is the "Reduce" step hidden in the code?

Let's measure time with the following function (not optimal)

```
measure.time <- function(command)
{
  start.time <- Sys.time()
  eval(parse(text = command))
  end.time <- Sys.time()
  d <- difftime(end.time, start.time
, units = "secs"))
  return(d)
}
# Example:
# cmd <- "sum(parallel.loss.dist(1, 1e+08))"
# measure.time(cmd)
```

play with N to see when it pays off to use parallel or non-parallel version

adjust additionally the number of cores - does it get faster?

Thank you for your attention

You can find the presentation and the code that was used here at

`github.com/maxlit/workshops/tree/master/R/r-parallel-co`