Parallel Computation

02-04-2020

```
library(tidyverse)
```

```
## -- Attaching packages --------

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts -----------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Parallel Computation

Implict Parallel

- BLAS (Basic Linear Algebra Subroutines)
  - CRAN R shippings with a version of single threaded BLAS library.
  - Microsoft R Open ships with Interl MKL (Win/Linux) / Accelerate ML (macOS) BLAS libraries.
  - on macOS, R could be configured to use the optimized BLAS from Apple's Accelerate framework
  - We could only install R with different BLAS libraries such as openblas or ATLAS

Embarrassingly Parallel

Also called perfectly parallel, delightfully parallel or pleasingly parallel.

> An embarrassingly parallel task can be considered a trivial case - little or no manipulation is needed to separate the problem into a number of parallel tasks.

A bit deroute first - revisit some of our old friends `map` and `map_*` in `purrr`.

```
1:4 %>% map(function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
```

```
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
```

```
1:4 %>% map_dbl(function(x) x^2)
```

```
## [1]  1  4  9 16
```

These are the base R equivalence.

```
1:4 %>% lapply(function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
```

```
1:4 %>% sapply(function(x) x^2)
```

```
## [1]  1  4  9 16
```

Suppose we have a list of vectors and we want to operation some operation on each vector.

```
# it is a slow operation, imagine that in real applications, it could take a few minutes
slow_task <- function(x) {
  sum(x %o% x)
}

list_of_vectors <- replicate(10, list(rnorm(5000)))
list_of_vectors %>% glimpse()
```

```
## List of 10
##  $ : num [1:5000] 0.672 0.404 -0.909 -0.9 -0.18 ...
##  $ : num [1:5000] -0.0194 0.7694 -1.5328 -0.4778 1.3849 ...
##  $ : num [1:5000] 0.346 1.396 -0.318 0.631 1.331 ...
##  $ : num [1:5000] -0.521 0.126 -1.571 0.499 -0.187 ...
##  $ : num [1:5000] -0.293 -0.293 -1.325 0.618 -2.125 ...
##  $ : num [1:5000] -0.19 -1.657 -1.066 0.355 0.618 ...
##  $ : num [1:5000] 0.5755 -0.0787 0.6859 0.161 2.6518 ...
##  $ : num [1:5000] 0.891 0.404 -1.775 -0.19 0.529 ...
##  $ : num [1:5000] 0.0637 -0.1476 -0.596 0.6737 -0.7635 ...
##  $ : num [1:5000] 0.1735 -0.0299 -0.5185 -0.3936 0.764 ...
```

```
list_of_vectors %>% map_dbl(slow_task)
```

However, these commands only run in a single process, it means, if the list is doubled, the time is also at least doubled.

```
system.time({
  list_of_vectors %>% map_dbl(slow_task)
})
```

```
##    user  system elapsed
##   4.301   1.966   6.813
```

```
# double the list
longer_list_of_vectors <- c(list_of_vectors, list_of_vectors)
system.time({
  longer_list_of_vectors %>% map_dbl(slow_task)
})
```

```
##    user  system elapsed
##   7.732   3.822  12.508
```

We are hoping to use multiple processes to speed up the job. The traditional way is to use the `parallel` package.

The package `parallel`

```
library(parallel)
```

Consider again the above list_vector example,

```
# the number of cores we have
detectCores()
```

```
## [1] 8
```

```
# it will create a socket cluster on my own computer
cl <- makeCluster(4)
parLapply(cl, list_of_vectors, slow_task)
```

```
## [[1]]
## [1] 91.08713
##
## [[2]]
## [1] 116.0685
##
## [[3]]
## [1] 1.577384
##
```

```
## [[4]]
## [1] 21503.58
##
## [[5]]
## [1] 17554.92
##
## [[6]]
## [1] 19.08825
##
## [[7]]
## [1] 33273.63
##
## [[8]]
## [1] 153.5831
##
## [[9]]
## [1] 0.7945499
##
## [[10]]
## [1] 406.6671
```

```
# or if you want simplified result
parSapply(cl, list_of_vectors, slow_task)
```

```
##  [1] 9.108713e+01 1.160685e+02 1.577384e+00 2.150358e+04 1.755492e+04
##  [6] 1.908825e+01 3.327363e+04 1.535831e+02 7.945499e-01 4.066671e+02
```

```
# stop the cluster after use
stopCluster(cl)
```

Remark: you don't have to make and stop clusters for every operation, you could make a cluster in the very beginning of your script and close it at the very end.

Let's test the speed improvement

```
sum_each <- function(x, fun, n_cores) {
  cl <- makeCluster(n_cores)
  result <- parLapply(cl, x, fun)
  stopCluster(cl)
  result
}

system.time(sum_each(longer_list_of_vectors, slow_task, 2))
```

```
##    user  system elapsed
##   0.027   0.024   7.505
```

```
system.time(sum_each(longer_list_of_vectors, slow_task, 3))
```

```
##    user  system elapsed
##   0.033   0.024   5.520
```

```
system.time(sum_each(longer_list_of_vectors, slow_task, 4))
```

```
##    user  system elapsed
##   0.029   0.029   5.496
```

Processing Chunk

The iteratable is divided into chunks before sending the chunks to the workers. `Sys.getpid()` tells us the process id of a worker.

```
cl <- makeCluster(4)
```

```
parSapply(cl, 1:10, function(x) {
    Sys.getpid()
})
```

```
##  [1] 99297 99297 99297 99316 99316 99329 99329 99341 99341 99341
```

```
parSapply(cl, 1:10, function(x) {
    Sys.getpid()
  },
  chunk.size = 2
)
```

```
##  [1] 99297 99297 99316 99316 99329 99329 99341 99341 99297 99297
```

```
parSapply(cl, 1:10, function(x) {
    Sys.getpid()
  },
  chunk.size = 1
)
```

```
##  [1] 99297 99316 99329 99341 99297 99316 99329 99341 99297 99316
```

```
stopCluster(cl)
```

Load balancing

`parLapply` pre-schedules the tasks to each work. It could be suboptimal when different tasks require different amount of time to complete.

```
cl <- makeCluster(4)
```

```
x <- c(3, 3, 1, 1, 1, 1, 1, 1)
pause <- function(x) {
  Sys.sleep(x)
}

system.time({
  parLapply(cl, x, pause, chunk.size = 2)
})
```

```
##    user  system elapsed
##   0.006   0.002   6.090
```

```
system.time({
  parLapply(cl, x, pause, chunk.size = 1)
})
```

```
##    user  system elapsed
##   0.007   0.002   4.012
```

Instead of preshceduling the tasks, a task could also be assigned to a free worker dynamically using `parLapplyLB`.

```
system.time({
  parLapplyLB(cl, x, pause)
})
```

```
##    user  system elapsed
##   0.011   0.002   3.015
```

Note that it only takes 3 seconds now.

```
stopCluster(cl)
```

Interact directly with the workers

We just saw an quick example on using `parLapply`. Let's try a few more things.

```
cl <- makeCluster(4)
```

We could run some arbitrary commands on each of the workers

```
clusterEvalQ(cl, {
  x <- rnorm(100)
  mean(x)
})
```

```
## [[1]]
## [1] 0.1131698
##
## [[2]]
## [1] -0.09628778
##
## [[3]]
## [1] -0.05418027
##
## [[4]]
## [1] 0.08650927
```

```
clusterEvalQ(cl, {
  Sys.getpid()
})
```

```
## [[1]]
## [1] 99456
##
## [[2]]
## [1] 99479
##
## [[3]]
## [1] 99491
##
## [[4]]
## [1] 99503
```

Global variables in master are not exported to the worker automatically

```
y <- 3
clusterEvalQ(cl, {
  y + 1
})
```

```
## Error in checkForRemoteErrors(lapply(cl, recvResult)): 4 nodes produced errors; first error: object
```

`clusterExport` exports the global variables to each worker.

```
clusterExport(cl, "y")
clusterEvalQ(cl, {
  y + 1
})
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 4
```

If you want to set a random seed, the following doesn't work because each work returns the same result.

```
clusterEvalQ(cl, {
  set.seed(123)
  rnorm(5)
})
```

```
## [[1]]
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##
## [[2]]
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##
## [[3]]
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##
## [[4]]
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

```
clusterSetRNGStream(cl, 123)
clusterEvalQ(cl, {
  rnorm(5)
})
```

```
## [[1]]
## [1] -0.9685927  0.7061091  1.4890213 -1.8150926  0.3304096
##
## [[2]]
## [1] -0.4094454  0.8909694 -0.8653704  1.4642711  1.2674845
##
## [[3]]
## [1] -0.48906078  0.43304237 -0.03195349  0.14670372 -1.75239095
##
## [[4]]
## [1] -1.0388664  1.5745125  0.7470820  0.6718720  0.2691436
```

```
# do not forget to close the cluster
stopCluster(cl)
```

`map` or `lapply` like syntax

- `mclapply` from `parallel` (unix / macOS only)

Remark: `mclapply` relies on forking, it means that it doesn't work on Windows. We will discuss a cross platform approach.)

```
list_of_vectors %>% mclapply(
  slow_task,
  mc.cores = 4
)
```

```
## [[1]]
## [1] 91.08713
##
## [[2]]
## [1] 116.0685
##
## [[3]]
```

```
## [1] 1.577384
##
## [[4]]
## [1] 21503.58
##
## [[5]]
## [1] 17554.92
##
## [[6]]
## [1] 19.08825
##
## [[7]]
## [1] 33273.63
##
## [[8]]
## [1] 153.5831
##
## [[9]]
## [1] 0.7945499
##
## [[10]]
## [1] 406.6671
```

```
list_of_vectors %>% mclapply(
  slow_task,
  mc.preschedule = FALSE,  #  set FALSE to enable load balancing
  mc.cores = 4
)
```

```
## [[1]]
## [1] 91.08713
##
## [[2]]
## [1] 116.0685
##
## [[3]]
## [1] 1.577384
##
## [[4]]
## [1] 21503.58
##
## [[5]]
## [1] 17554.92
##
## [[6]]
## [1] 19.08825
##
## [[7]]
## [1] 33273.63
##
## [[8]]
## [1] 153.5831
##
## [[9]]
```

```
## [1] 0.7945499
##
## [[10]]
## [1] 406.6671
```

- package `furrr`

`furrr` provides functions which are very similar to those in `purrr`.

```
library(furrr)
```

```
## Loading required package: future
```

```
plan(multiprocess, workers = 4)
system.time({
  future_map(c(2, 2, 2, 2), ~Sys.sleep(.))
})
```

```
##    user  system elapsed
##   0.199   0.009   2.824
```

```
future_map_dbl(list(1:10, 11:20, 21:30, 31:41), ~ sum(.))
```

```
## [1]  55 155 255 396
```

How to do load balanacing with `future_map`?

```
# without load balanacing
system.time({
  future_map(
    c(3, 3, 1, 1, 1, 1, 1, 1),
    ~Sys.sleep(.))
})
```

```
##    user  system elapsed
##   0.261   0.021   6.096
```

```
# with load balanacing
system.time({
  future_map(
    c(3, 3, 1, 1, 1, 1, 1, 1),
    ~Sys.sleep(.),
    .options = future_options(scheduling = FALSE))
})
```

```
##    user  system elapsed
##   0.508   0.023   4.170
```

foreach

Reference:

- R Programming for Data Science https://bookdown.org/rdpeng/rprogdatascience/parallel-computation.html