Parallel Computation

02-04-2020

```
library(tidyverse)
```

```
## -- Attaching packages ------ tidyverse 1.3.0 --

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts --------- tidyverse_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 Intel 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
```

```
1:4 %>% map_dbl(~ .^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] 0.922 1.916 1.554 -1.3 -0.728 ...
##  $ : num [1:5000] -1.347 0.212 0.635 0.51 0.421 ...
##  $ : num [1:5000] 0.9956 0.0839 0.1026 1.2007 1.696 ...
##  $ : num [1:5000] -0.794 0.752 0.66 1.458 -0.202 ...
```

```
##  $ : num [1:5000] 1.067 0.919 0.338 -1.238 0.853 ...
##  $ : num [1:5000] -0.5488 0.5762 0.9068 0.8515 -0.0342 ...
##  $ : num [1:5000] -0.8625 -0.876 0.0887 -0.5556 -0.9282 ...
##  $ : num [1:5000] -0.521 -0.864 1.265 -0.21 -1.203 ...
##  $ : num [1:5000] 1.802 -1.218 -0.44 0.828 -1.354 ...
##  $ : num [1:5000] -1.277 0.449 1.292 -0.552 0.737 ...
```

```
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
##   1.639   0.707   3.075
```

```
# 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
##   2.925   1.409   5.002
```

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] 4
```

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

```
## [[1]]
## [1] 4811.179
```

```
##
## [[2]]
## [1] 1459.21
##
## [[3]]
## [1] 1552.101
##
## [[4]]
## [1] 686.962
##
## [[5]]
## [1] 29850.26
##
## [[6]]
## [1] 4672.394
##
## [[7]]
## [1] 11657.81
##
## [[8]]
## [1] 3712.043
##
## [[9]]
## [1] 3964.806
##
## [[10]]
## [1] 6.177521
```

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

```
##  [1]  4811.179307  1459.210373  1552.101414   686.962010 29850.264870
##  [6]  4672.394230 11657.813928  3712.043070  3964.805689     6.177521
```

```
# 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

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

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

```
##    user  system elapsed
##   0.006   0.006   2.740
```

```
system.time(run_each(longer_list_of_vectors, slow_task, 8))
```

```
##    user  system elapsed
##   0.020   0.019   3.654
```

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] 91359 91359 91359 91378 91378 91390 91390 91402 91402 91402
```

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

```
##  [1] 91359 91359 91378 91378 91390 91390 91402 91402 91359 91359
```

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

```
##  [1] 91359 91378 91390 91402 91359 91378 91390 91402 91359 91378
```

```
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({
  map(x, pause)
})
```

```
##    user  system elapsed
##   0.002   0.000  12.013
```

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

```
##    user  system elapsed
##   0.001   0.000   6.003
```

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

```
##    user  system elapsed
##   0.001   0.000   6.004
```

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

```
##    user  system elapsed
##   0.002   0.001   4.011
```

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
##   0.003   0.000   3.013
```

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] 0.006329082
##
## [[2]]
## [1] 0.04122883
##
## [[3]]
## [1] -0.1234823
##
## [[4]]
## [1] -0.1595143
```

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

```
## [[1]]
## [1] 91492
##
## [[2]]
## [1] 91505
##
## [[3]]
## [1] 91517
##
## [[4]]
## [1] 91529
```

Global variables in master are not exported to the worker automatically (the same is true for `parLapply`)

```
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.

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

```
## [[1]]
## [1]  1.10581558  0.21022034 -0.06865181 -1.44096994 -0.03902856
##
## [[2]]
## [1] -1.4189434 -1.2871294 -1.1269572  0.6394698  1.0231722
##
## [[3]]
## [1] -0.4562880 -0.8421999 -1.2486595 -0.6615358  0.9060979
##
## [[4]]
## [1] -0.8552098  0.8159713 -1.8858255  0.9229573 -1.1162665
```

```
# wrong again
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)
```

Another pouplar package: `foreach`

As we have seen a bit earlier, we might need to use `clusterExport` to export certain gloabl variables. It would be a bit cumbersome. To reduce the extra steps, we could consider `foreach` and `doParallel`. They will send all the globals to the workers before running the tasks.

```
library(foreach)
```

```
##
## Attaching package: 'foreach'

## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
library(doParallel)
```

```
## Loading required package: iterators
```

```
cl <- makeCluster(4)
registerDoParallel(cl)
m <- matrix(rnorm(9), 3, 3)
# matrix m is sent to the workers implictly
# in fact, all globals are sent to the workers in default
foreach(i = seq_len(nrow(m)), .combine = c) %dopar% {
  sum(m[i, ])
}
```

```
## [1] -0.02905105 -1.36595099  2.58692045
```

```
stopCluster(cl)
```

However, it is not quite "functional" as `map`, `lapply` and `parLapply`.

`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.)

```r
# in default, `mclapply` uses 2 cores
system.time({
  mclapply(
    c(2, 2, 2, 2),
    function(x) Sys.sleep(x))
})
```

```
##    user  system elapsed
##   0.006   0.014   4.014
```

```r
system.time({
  mclapply(
    c(3, 3, 1, 1, 1, 1, 1, 1),
    function(x) Sys.sleep(x),
    mc.preschedule = FALSE,  #  set FALSE to enable load balancing
    mc.cores = 4
  )
})
```

```
##    user  system elapsed
##   0.012   0.017   3.016
```

package furrr

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

```r
library(furrr)
```

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

```r
# to use 4 workers, omit it to use all available cores
plan(multiprocess, workers = 4)
system.time({
  future_map(
    c(2, 2, 2, 2),
    ~Sys.sleep(.))
})
```

```
##    user  system elapsed
##   0.051   0.002   2.213
```

future_map has a family of type speific functions. For example,

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

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

load balanacing in future_map

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

```
##    user  system elapsed
##   0.061   0.002   6.026
```

```r
# 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
##   0.089   0.003   3.915
```

The honorable mention `multidplyr`

The package `multidplyr` is tidyverse solution to multiprocess data management. However, it is not released to CRAN yet, it means we need to install it from its github repo by using the package `remotes`.

```r
remotes::install_github("tidyverse/multidplyr")
```

Since it is still under developement, use it with caution!

Reference:

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