Parallel Computation

02-04-2020

```r
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`.

```r
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] -1.7992 -0.0641 -0.4177 2.8216 -1.1874 ...
##  $ : num [1:5000] 1.036 -0.21 -2.687 0.168 0.222 ...
##  $ : num [1:5000] 0.3148 0.7518 -0.0846 0.6298 0.464 ...
##  $ : num [1:5000] 1.427 1.045 1.081 0.502 0.737 ...
##  $ : num [1:5000] -0.889 0.948 -0.357 -1.646 -0.682 ...
##  $ : num [1:5000] 1.853 -1.853 0.201 0.883 0.418 ...
##  $ : num [1:5000] 1.7804 0.0976 -0.8106 0.3524 0.4525 ...
##  $ : num [1:5000] 0.07648 -0.15392 -0.27603 -0.6108 0.00352 ...
##  $ : num [1:5000] 0.541 -1.721 -0.272 -1.816 1.159 ...
##  $ : num [1:5000] -2.496 0.415 -1.033 0.286 -0.15 ...
```

```
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
##   1.682   0.807   3.031
```

```
# 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
##   2.958   1.508   5.027
```

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] 4155.309
```

```
##
## [[2]]
## [1] 170.4475
##
## [[3]]
## [1] 3331.136
##
## [[4]]
## [1] 366.7883
##
## [[5]]
## [1] 1670.017
##
## [[6]]
## [1] 458.2525
##
## [[7]]
## [1] 5573.779
##
## [[8]]
## [1] 270.4249
##
## [[9]]
## [1] 5203.904
##
## [[10]]
## [1] 487.8705
```

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

```
##  [1] 4155.3086  170.4475 3331.1358  366.7883 1670.0172  458.2525 5573.7787
##  [8]  270.4249 5203.9036  487.8705
```

```r
# 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

```r
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
##   0.005   0.005   2.588
```

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

```
##    user  system elapsed
##   0.021   0.019   4.612
```

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.

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

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

```
##  [1] 9956 9956 9956 9969 9969 9981 9981 9993 9993 9993
```

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

```
##  [1] 9956 9956 9969 9969 9981 9981 9993 9993 9956 9956
```

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

```
##  [1] 9956 9969 9981 9993 9956 9969 9981 9993 9956 9969
```

```r
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.

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

```r
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   0.001  12.014
```

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

```
##    user  system elapsed
##   0.001   0.000   6.007
```

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

```
##    user  system elapsed
##   0.001   0.000   6.005
```

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

```
##    user  system elapsed
##   0.002   0.001   4.008
```

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

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

```
##    user  system elapsed
##   0.003   0.001   3.011
```

Note that it only takes 3 seconds now.

```r
stopCluster(cl)
```

Interact directly with the workers

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

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

We could run some arbitrary commands on each of the workers

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

```
## [[1]]
## [1] -0.01966854
##
## [[2]]
## [1] 0.06790044
##
## [[3]]
## [1] 0.05676294
##
## [[4]]
## [1] -0.05707089
```

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

```
## [[1]]
## [1] 10084
##
## [[2]]
## [1] 10097
##
## [[3]]
## [1] 10110
##
## [[4]]
## [1] 10123
```

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.

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

```
## [[1]]
## [1] -0.31435101 -0.19035421 -0.72131909 -0.06728525 -0.45529078
##
## [[2]]
## [1] -0.8173919 -1.5920075  1.0788678 -0.6930334  0.4235804
##
## [[3]]
## [1]  0.5955447 -0.2060396  0.2499855 -1.4435495  0.1107902
##
## [[4]]
## [1]  1.0292018 -1.3022241 -0.1102215  0.4754966 -0.6773426
```

```r
# 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
```

```r
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
```

```r
# 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.

```r
library(foreach)
```

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

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

```r
library(doParallel)
```

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

```r
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
```

```r
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.)

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

```
##    user  system elapsed
##   0.004   0.007   4.025
```

```
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
##   0.014   0.023   3.022
```

package furrr

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

```
library(furrr)
```

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

```
# to use 4 workers, `plan(multiprocess)` will use all the avaliable workers
plan(multiprocess, workers = 4)
system.time({
  future_map(
    c(2, 2, 2, 2),
    ~ Sys.sleep(.)
  )
})
```

```
##    user  system elapsed
##   0.053   0.002   2.270
```

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

```
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
##   0.063   0.003   6.038
```

```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
##   0.092   0.005   3.926
```

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!

```r
library(multidplyr)
```

There are two ways to get data to the workers in cluster:

- `partition()` a data frame that already loaded in the interactive process.
- Load a different subset of the data in each worker.

Use `partition()` to send data to workers

```r
library(nycflights13)

cluster <- new_cluster(4)

flights %>%
  group_by(dest, origin) %>%
  partition(cluster) %>%
  summarize(air_time = mean(air_time, na.rm = TRUE))
```

```
## Source: party_df [224 x 3]
## Groups: dest
## Shards: 4 [43--67 rows]
##
##   dest  origin air_time
##   <chr> <chr>     <dbl>
## 1 ABQ   JFK       249.
## 2 ATL   JFK       112.
## 3 AVL   EWR        89.8
## 4 AVL   LGA        92.2
## 5 BDL   EWR        25.5
## 6 BHM   JFK       117
## # ... with 218 more rows
```

Load data in each worker

```r
cluster <- new_cluster(4)

cluster %>% cluster_library(c("tidyverse", "nycflights13"))

cluster %>%
  cluster_assign_partition(destination = unique(flights$dest))

# let's check the `destimation` variable in the workers
cluster %>%
  cluster_call(destination)
```

```
## [[1]]
##  [1] "IAH" "MIA" "BQN" "ATL" "ORD" "FLL" "IAD" "MCO" "PBI" "TPA" "LAX" "SFO"
## [13] "DFW" "BOS" "LAS" "MSP" "DTW" "RSW" "SJU" "PHX" "BWI" "CLT" "BUF" "DEN"
## [25] "SNA" "MSY" "SLC"
##
## [[2]]
##  [1] "XNA" "MKE" "SEA" "ROC" "SYR" "SRQ" "RDU" "CMH" "JAX" "CHS" "MEM" "PIT"
## [13] "SAN" "DCA" "CLE" "STL" "MYR" "JAC" "MDW" "HNL" "BNA" "AUS" "BTV" "PHL"
## [25] "STT" "EGE"
##
## [[3]]
##  [1] "AVL" "PWM" "IND" "SAV" "CAK" "HOU" "LGB" "DAY" "ALB" "BDL" "MHT" "MSN"
## [13] "GSO" "CVG" "BUR" "RIC" "GSP" "GRR" "MCI" "ORF" "SAT" "SDF" "PDX" "SJC"
## [25] "OMA" "CRW"
##
## [[4]]
##  [1] "OAK" "SMF" "TUL" "TYS" "OKC" "PVD" "DSM" "PSE" "BHM" "CAE" "HDN" "BZN"
## [13] "MTJ" "EYW" "PSP" "ACK" "BGR" "ABQ" "ILM" "MVY" "SBN" "LEX" "CHO" "TVC"
## [25] "ANC" "LGA"
```

```r
cluster %>% cluster_send({
  df <- flights %>% filter(dest %in% destination)
})

cluster %>%
  party_df("df") %>%
  group_by(dest, origin) %>%
  summarize(air_time = mean(air_time, na.rm = TRUE)) %>%
  collect()
```

```
## # A tibble: 224 x 3
## # Groups:   dest [105]
##    dest  origin air_time
##    <chr> <chr>     <dbl>
##  1 ATL   EWR       112.
##  2 ATL   JFK       112.
##  3 ATL   LGA       114.
##  4 BOS   EWR        40.3
##  5 BOS   JFK        38.5
##  6 BOS   LGA        37.9
##  7 BQN   EWR       196.
##  8 BQN   JFK       194.
##  9 BUF   EWR        51.2
## 10 BUF   JFK        57.1
## # ... with 214 more rows
```

Reference:

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