Parallelization and Profiling

Programming for Statistical Science

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- Getting Started with `doMC` and `foreach` vignette
- `profvis` guide

Recall

Benchmarking with package bench

```
library(bench)
x <- runif(n = 1000000)
bench::mark(
  sqrt(x),
  x ^ 0.5,
  x ^ (1 / 2),
  min_time = Inf, max_iterations = 1000
)
```

```
#> # A tibble: 3 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 sqrt(x)      2.12ms   2.37ms     398.     7.63MB   115.
#> 2 x^0.5       17.12ms  19.13ms      52.6    7.63MB     8.70
#> 3 x^(1/2)     17.22ms  18.48ms      53.8    7.63MB     8.99
```

Functions `Sys.time()` and `bench::system_time()` are also available for you to time your code.

Ways to parallelize

1. Sockets

   A new version of R is launched on each core.
   - Available on all systems
   - Each process on each core is unique

2. Forking

   A copy of the current R session is moved to new cores.
   - **Not available on Windows**
   - Less overhead and easy to implement

Package `parallel`

```
library(parallel)
```

Some core functions:

- `detectCores()`
- `pvec()`, parallelize a vector map function using forking
  - Argument `mc.cores` is used to set the number of cores
- `mclapply()`, parallel version of `lapply()` using forking
  - Argument `mc.cores` is used to set the number of cores
  - Arguments `mc.preschedule` and `affinity.list` can be used for load balancing.
- `mcparallel()`, `mccollect()`, evaluate an R expression asynchronously in a separate process

Our DSS R cluster has 16 cores available for use while your laptop probably has 4 or 8.

Load balancing example

Recall: `mclapply()` relies on forking.

```
sleepR <- function(x) {
  Sys.sleep(x)
  runif(1)
}
x <- c(2.5, 2.5, 5)
aff_list_bal <- c(1, 1, 2)
aff_list_unbal <- c(1, 2, 2)
```

```
# balanced load
system.time({
  mclapply(x, sleepR, mc.cores = 2,
           mc.preschedule = FALSE, affinity.list = aff_list_bal)
})
```

```
#>    user  system elapsed
#>   0.008   0.010   5.019
```

```
# unbalanced load
system.time({
  mclapply(x, sleepR, mc.cores = 2,
           mc.preschedule = FALSE, affinity.list = aff_list_unbal)
})
```

```
#>    user  system elapsed
#>   0.007   0.009   7.516
```

Sockets

Using sockets to parallelize

The basic recipe is as follows:

```
detectCores()
c1 <- makeCluster()
result <- clusterApply(cl = c1, ...)
stopCluster(c1)
```

Here you are spawning new R sessions. Data, packages, functions, etc. need to be shipped to the workers.

Sockets example

Function `clusterEvalQ()` evaluates a literal expression on each cluster node.

```
clust <- makeCluster(4)
library(nycflights13)
clusterEvalQ(cl = clust, dim(flights))
stopCluster(clust)
```

```
Error in checkForRemoteErrors(lapply(cl, recvResult)) :
  4 nodes produced errors; first error: object 'flights' not found
```

There is no inheritance. Package `nycflights13` is not loaded on the new R sessions spawned on each individual core.

```
clust <- makeCluster(4)
clusterEvalQ(cl = clust, {
                library(nycflights13)
                dim(flights)})
```

```
#> [[1]]
#> [1] 336776     19
#>
#> [[2]]
#> [1] 336776     19
#>
#> [[3]]
#> [1] 336776     19
#>
#> [[4]]
#> [1] 336776     19
```

```
stopCluster(clust)
```

Function `clusterExport()` can be used to pass objects from the master process to the corresponding spawned sessions.

```
cl <- makeCluster(4)
library(nycflights13)
clusterExport(cl = cl, varlist = c("flights"))
clusterEvalQ(cl = cl, {dim(flights)})
```

```
#> [[1]]
#> [1] 336776     19
#>
#> [[2]]
#> [1] 336776     19
#>
#> [[3]]
#> [1] 336776     19
#>
#> [[4]]
#> [1] 336776     19
```

```
stopCluster(cl)
```

Apply operations using clusters

There exists a family of analogous apply functions that use clusters.

Function	Description
`parApply()`	parallel version of `apply()`
`parLapply()`	parallel version of `lapply()`
`parLapplyLB()`	load balancing version of `parLapply()`
`parSapply()`	parallel version of `sapply()`
`parSapplyLB()`	load balancing version of `parSapply()`

The first argument is a cluster object. Subsequent arguments are similar to the corresponding base `apply()` variants.

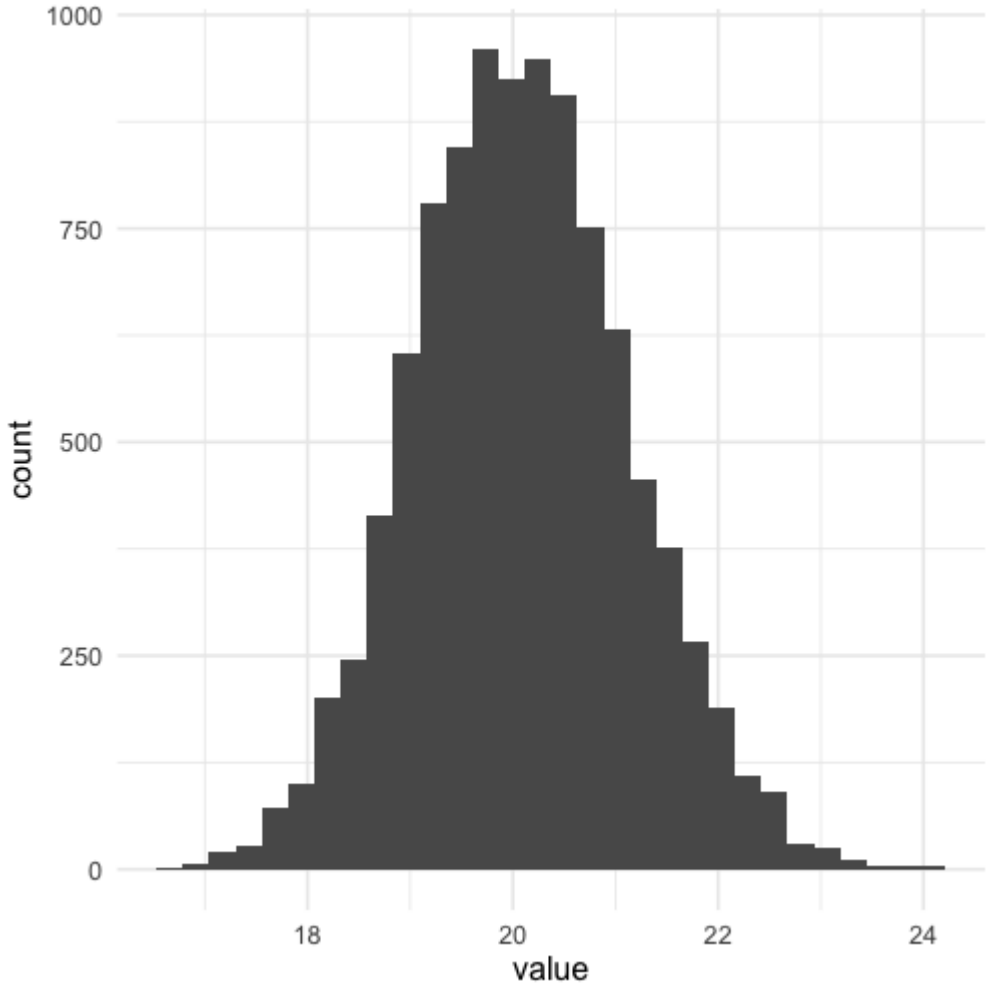
Bootstrapping

Parallelize the bootstrap process using `dplyr` functions.

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

boot_samples <- clusterEvalQ(cl = cl, {
    library(dplyr)
    create_boot_sample <- function() {
      mtcars %>%
        select(mpg) %>%
        sample_n(size = nrow(mtcars), replace = TRUE)
    }
    replicate(2500, create_boot_sample())
  }
)
```

```
map(boot_samples, ~parLapply(cl, X = ., fun = mean)) %>%
  unlist() %>%
  as_tibble() %>%
  ggplot(aes(x = value)) +
  geom_histogram() +
  theme_minimal(base_size = 16)
```




```
stopCluster(cl)
```

doMC **and** foreach

Parallelized `for` loop

Package `doMC` is a parallel backend for the `foreach` package - a package that allows you to execute `for` loops in parallel.

```
library(doMC)
library(foreach)
```

Key functions:

- `doMC::registerDoMC()`, set the number of cores for the parallel backend to be used with `foreach`
- `foreach`, `%dopar%`, `%do%`, parallel loop

`doMC` serves as an interface between `foreach` and `multicore`. Since `multicore` only works with systems that support forking, these functions will not work properly on Windows.

Set workers

To get started, set the number of cores with `registerDoMC()`.

```
# check cores set up
getDoParWorkers()
```

```
#> [1] 1
```

```
# set 4 cores
registerDoMC(4)
getDoParWorkers()
```

```
#> [1] 4
```

Serial and parallel with `foreach()`

Sequential

```
foreach(i = 1:4) %do%
  sort(runif(n = 1e7, max = i))[1]
```

```
#> [[1]]
#> [1] 1.043081e-07
#>
#> [[2]]
#> [1] 1.625158e-07
#>
#> [[3]]
#> [1] 4.470348e-08
#>
#> [[4]]
#> [1] 9.313226e-09
```

```
times(2) %do%
  sort(runif(n = 1e7))[1]
```

```
#> [1] 2.093147e-07 1.059379e-07
```

Parallel

```
foreach(i = 1:4) %dopar%
  sort(runif(n = 1e7, max = i))[1]
```

```
#> [[1]]
#> [1] 1.44355e-08
#>
#> [[2]]
#> [1] 6.798655e-08
#>
#> [[3]]
#> [1] 9.848736e-08
#>
#> [[4]]
#> [1] 1.490116e-08
```

```
times(2) %dopar%
  sort(runif(n = 1e7))[1]
```

```
#> [1] 4.251488e-07 4.237518e-08
```

Time comparison

Sequential

```
system.time({
  foreach(i = 1:4) %do%
    sort(runif(n = 1e7, max = i))[1]
})
```

```
#>    user  system elapsed
#>   3.296   0.144   3.448
```

```
system.time({
  for (i in 1:4)
    sort(runif(n = 1e7, max = i))[1]
})
```

```
#>    user  system elapsed
#>   3.472   0.107   3.589
```

Parallel

```
system.time({
  foreach(i = 1:4) %dopar%
    sort(runif(n = 1e7, max = i))[1]
})
```

```
#>    user  system elapsed
#>   2.453   0.335   1.440
```

Even with four cores we don't see a four factor improvement in time.

Iterate over multiple indices

Add more indices separated by commas. Argument `.combine` allows you to format the result into something other than the default list.

Equal `i` and `j`

```
foreach(i = 1:3, j = -2:0, .combine = "c") %dopar% {i ^ j}
```

```
#> [1] 1.0 0.5 1.0
```

Longer `j`

```
foreach(i = 1:3, j = -3:0, .combine = "c") %dopar% {i ^ j}
```

```
#> [1] 1.0000000 0.2500000 0.3333333
```

Longer `i`

```
foreach(i = 1:4, j = 0:1, .combine = "c") %dopar% {i ^ j}
```

```
#> [1] 1 2
```

Length coercion is not supported. We'll need a nested structure.

Nested `foreach` loops

The `%:%` operator is the nesting operator, used for creating nested `foreach` loops.

```
foreach(i = 1:4, .combine = "c") %:%
  foreach(j = 0:1, .combine = "c") %dopar%
    {i ^ j}
```

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

```
foreach(i = 1:4, .combine = "data.frame") %:%
  foreach(j = 0:1, .combine = "c") %dopar%
    {i ^ j}
```

```
#>   result.1 result.2 result.3 result.4
#> 1        1        1        1        1
#> 2        1        2        3        4
```

```
foreach(i = 1:4, .combine = "c") %:%
  foreach(j = 0:1, .combine = "+") %dopar%
    {i ^ j}
```

```
#> [1] 2 3 4 5
```

Exercise

The 1986 crash of the space shuttle Challenger was linked to failure of O-ring seals in the rocket engines. Data was collected on the 23 previous shuttle missions.

Perform leave-one-out cross validation in parallel fitting a logistic regression model where the response is `damage` / `no_damage`, predictor is `temp`, and data is from `orings` in package `faraway`.

```
library(tidyverse)
library(faraway)
data("orings")
orings_logistic <- orings %>%
  mutate(damage = ifelse(damage > 0, 1, 0))
```

Compute the average test errors:

$$\text{average test error} = \frac{1}{n}\sum_{i=1}^{n} 1_{(y_i \neq \hat{y}_i^{-i})}$$

Exercise hint

Perform leave-one-out cross validation in parallel fitting a logistic regression model where the response is `damage` / `no_damage`, predictor is `temp`, and data is from `orings` in package `faraway`.

```
library(tidyverse)
library(faraway)
data("orings")
orings_logistic <- orings %>%
  mutate(damage = ifelse(damage > 0, 1, 0))
```

Compute the average test errors:

$$\text{average test error} = \frac{1}{n}\sum_{i=1}^{n} 1_{(y_i \neq \hat{y}_i^{-i})}$$

Template code:

```
m <- glm(damage ~ temp, family = "binomial",
           data = orings_logistic[-i, , drop = FALSE])
y_hat <- round(predict(m, newdata = orings_logistic[i, , drop = FALSE],
                       type = "response"))
y <- orings_logistic[i, , drop = FALSE]$damage
```

More bootstrap

Create a function that returns $\hat{\beta}_1$.

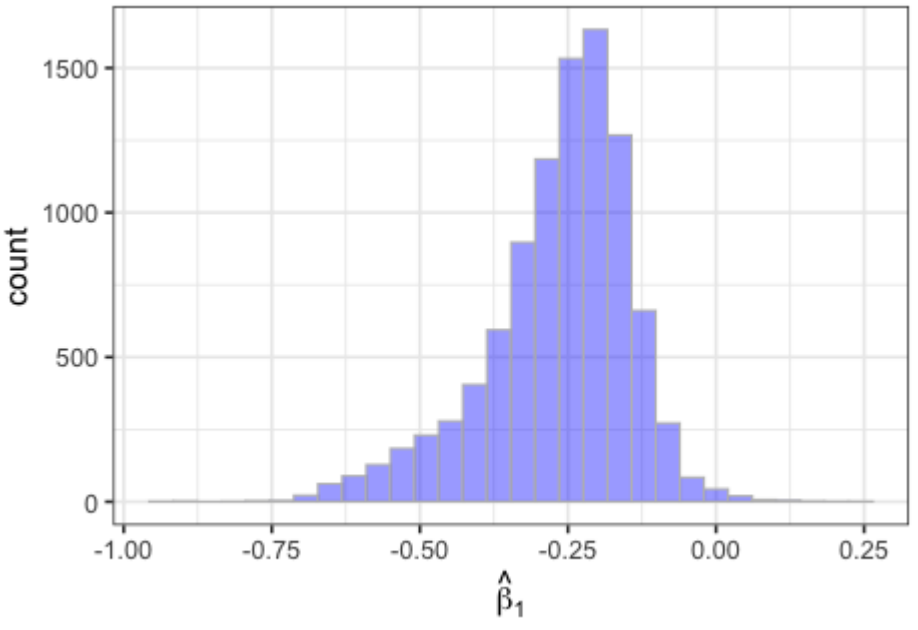
```
quiet_glm <- quietly(glm)

get_b1 <- function() {
  orings_boot <- orings_logistic %>%
    sample_n(size = dim(orings_logistic)[1], replace = TRUE)
  m <- quiet_glm(damage ~ temp,
                 family = "binomial", data = orings_boot)
  b1 <- coef(m$result)["temp"]
  if (length(m$warnings)) {b1 <- NULL}
  return(b1)
}
```

Generate 10,000 bootstrap samples.

```
N <- 10000
registerDoMC(4)
b1_boot_sample <- times(N) %dopar% get_b1()
```

```
tibble(x = b1_boot_sample) %>%
  ggplot(aes(x = x)) +
  geom_histogram(bins = 30, fill = "blue", color = "grey", alpha = .4) +
  labs(x = expression(hat(beta)[1])) + theme_bw(base_size = 16)
```




```
quantile(b1_boot_sample, c(.025, .975))
```

```
#>       2.5%      97.5%
#> -0.5726802 -0.0763743
```

```
quantile(b1_boot_sample, c(.03, .98))
```

```
#>          3%         98%
#> -0.55545004 -0.06787193
```

Time check

In parallel, 4 cores:

```
N <- 10000
registerDoMC(4)
system.time({b1_boot_sample <- times(N) %dopar% get_b1()})
```

```
#>    user  system elapsed
#>  29.540   3.131   8.943
```

In parallel, 8 cores:

```
registerDoMC(8)
system.time({b1_boot_sample <- times(N) %dopar% get_b1()})
```

```
#>    user  system elapsed
#>  39.809   4.487   7.929
```

Sequentially:

```
system.time({replicate(N, get_b1())})
```

```
#>    user  system elapsed
#>  18.921   1.559  20.562
```

Profiling

Profiling with `profvis`

We can do more than just time our code. Package `profvis` provides an interactive interface to visualize profiling data.

```
library(profvis)
```

To profile your code

- wrap your R expression inside `profvis()`,
- or use RStudio's GUI under the `Profile` tab.

Exercise

First, profile the below code. Then, try to improve the computation time while keeping the loops and not using parallel computing. Lastly, try an apply variant and evaluate the performance.

```
reps <- 10000
n <- 1000

beta_0 <- 2
beta_1 <- .5
beta_2 <- 3

beta_1_hat_all <- c()

for (s in c(1, 3, 7)) {
  beta_1_hat <- c()
  for (i in 1:reps) {
    X <- cbind(rnorm(n), rnorm(n) ^ 2)
    Y <- beta_0 + beta_1 * X[, 1, drop = FALSE]  +
      beta_2 * X[, 2, drop = FALSE] + rnorm(n, sd = s)
    m <- lm(Y~X)
    beta_1_hat <- c(beta_1_hat, coefficients(m)[2])
  }
  beta_1_hat_all <- c(beta_1_hat_all, beta_1_hat)
}

beta_df <- tibble(sd         = rep(c(1, 3, 7), each = reps),
                  beta_1_hat = beta_1_hat_all)
```

Save profile

```
library(profvis)
p <- profvis({reps <- 10000
n <- 1000

beta_0 <- 2
beta_1 <- .5
beta_2 <- 3

beta_1_hat_all <- c()

for (s in c(1, 3, 7)) {
  beta_1_hat <- c()
  for (i in 1:reps) {
    X <- cbind(rnorm(n), rnorm(n) ^ 2)
    Y <- beta_0 + beta_1 * X[, 1, drop = FALSE]  +
      beta_2 * X[, 2, drop = FALSE] + rnorm(n, sd = s)
    m <- lm(Y~X)
    beta_1_hat <- c(beta_1_hat, coefficients(m)[2])
  }
  beta_1_hat_all <- c(beta_1_hat_all, beta_1_hat)
}

beta_df <- tibble(sd          = rep(c(1, 3, 7), each = reps),
                  beta_1_hat = beta_1_hat_all)})

htmlwidgets::saveWidget(p, "profile.html")
```

Profiled code

Tips for improving performance

1. Identify bottlenecks in your code - you have to know what code to focus on.

2. Slim down your functions. Use a specific function for a specific problem.

    - Do you need everything that comes with the output of `lm()`?
    - Do you only want the p-values from 1,000 tests?

3. Vectorise

    - Matrix algebra is a form of vectorization. The loops are executed via external libraries such as BLAS.

4. Avoid copies

    - Be cautious with `c()`, `append()`, `cbind()`, `rbind()`, or `paste()`.
    - Check how often the garbage collector is running in your profiled code.

References

1. Profvis — Interactive Visualizations for Profiling R Code. (2020). https://rstudio.github.io/profvis/.

2. Weston, Steve. Getting started with doMC and foreach. (2020). https://cran.r-project.org/web/packages/doMC/vignettes/gettingstartedMC.pdf