Computational Skills for Biostatistics I: Lecture 5

Amy Willis, Biostatistics, UW

01 May, 2019

Theme for today: Computation time

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered."

— Donald Knuth

Theme for today: Computation time

"R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough."

— (BFF) Hadley Wickham

Theme for today: Computation time

A language that was built for data analysis: R

- Of course you can do scientific computing in R

Languages that were built for scientific computing: Python, MatLab, C++

- Of course you can do data analysis in these languages

Theme for today: Computation time

However, with some understanding of how R works, and some understanding of how to use your computer effectively, you can significantly speed up runtime.

Evaluating runtime

You can grab the "system time": how long have you been running your R process

```
proc.time()
```

```
##    user  system elapsed
##   1.239   0.080   1.329
```

```
t1 <- proc.time()
Sys.sleep(0.25)
t2 <- proc.time()
t2 - t1
```

```
##    user  system elapsed
##   0.002   0.000   0.256
```

Evaluating runtime

You can also use `system.time()`

```
system.time(rnorm(1e5))
```

```
##    user  system elapsed
##   0.005   0.001   0.005
```

```
system.time(rnorm(1e7))
```

```
##    user  system elapsed
##   0.524   0.019   0.582
```

Evaluating runtime

Investigating chunks of code

```
t1 <- Sys.time()
my_inverse <- rnorm(1e6) %>% matrix(nrow = 1e3) %>% solve
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 0.9289382 secs
```

Microbenchmarking

For very small comparisons, the library `microbenchmark` is great!

```
library(microbenchmark)
x <- runif(100)
mbm <- microbenchmark(
  sqrt(x),
  x ^ 0.5
)
mbm
```

```
## Unit: nanoseconds
##      expr  min      lq     mean median      uq  max neval
##   sqrt(x)  249   263.5  402.63  276.0   323.0 4564   100
##    x^0.5 1877 1908.0 2107.60 1925.5 1971.5 6672   100
```

Microbenchmark

```
mbm <- microbenchmark(
  sqrt(x),
  x ^ 0.5
)
autoplot(mbm)
```

```
## Coordinate system already present. Adding new coordinate
```

Microbenchmarking

```
mbm
```

```
## Unit: nanoseconds
##       expr  min   lq     mean median     uq  max neval
##   sqrt(x)  256  267  298.92  280.5  309.5  780   100
##     x^0.5 1842 1869 1955.90 1878.5 1926.5 6894   100
```

Which should you use?

`sqrt()` or `^0.5`: Which should you use?

```
mbm
```

```
## Unit: nanoseconds
##      expr  min   lq    mean median     uq  max neval
##  sqrt(x)  256  267  298.92  280.5  309.5  780   100
##    x^0.5 1842 1869 1955.90 1878.5 1926.5 6894   100
```

It actually doesn't matter – a million square roots will take 0.5 or 2 seconds.

Don't agonise over microbenchmarks: no need to overoptimise

Tools for benchmarking

- `microbenchmark()`
- `Sys.time()`

Making larger gains in programming time

Start with the biggest bottleneck, work to speed it up as fast as possible; move onto the next biggest bottleneck, and so on...

There are easy and there are hard ways to make code run faster. Start with the easy ways!

The easiest way: run code in parallel

Tools for running code in parallel

- Writing functions: `parallel` and `snow`
- Data analysis: `multidplyr`
- Simulations: `simulator`

Quite practical example

An expensive operation to perform repeatedly is matrix inversion: $O(n^3)$

- I'm hoping we'll have time to come back to this notation

```
my_matrices <- replicate(n=20,
                         rnorm(1e2) %>% matrix(nrow = 1e1),
                         simplify=FALSE)
```

Let's distribute the work multiple cores

parallel

- The `parallel` package is an easy way to split computation over multiple cores

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

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

```
my_matrix_inverses <- mclapply(my_matrices,
                               solve,
                               mc.cores=4)
```

parallel

```
microbenchmark(times = 10,
  mclapply(my_matrices, solve, mc.cores=4),
  mclapply(my_matrices, solve, mc.cores=2),
  mclapply(my_matrices, solve, mc.cores=1)
)
```

```
## Unit: microseconds
##                                                expr      min
##  mclapply(my_matrices, solve, mc.cores = 4) 8021.691 872
##  mclapply(my_matrices, solve, mc.cores = 2) 4896.039 547
##  mclapply(my_matrices, solve, mc.cores = 1)  269.643  38
##     median         uq       max neval
##  9521.8155 10183.352 18095.226    10
##  6915.5480  7647.442  8174.207    10
##   624.2825  2029.152  2876.631    10
```

That's weird...

`mclapply`, `mcsapply`...

`mclapply`, `mcsapply` and friends use "forks"

- Idea from Unix-based systems
  - Does not work on Windows
- Takes a complete copy of the master process, including the workspace and state of the random-number stream
  - Generally fast but there is overhead

parallel

```
my_matrices <- replicate(n=20,
                         rnorm(1e2) %>% matrix(nrow = 1e1),
                         simplify=FALSE)
more_matrices <- replicate(n=20000,
                         rnorm(1e2) %>% matrix(nrow = 1e1),
                         simplify=FALSE)
```

parallel

```
microbenchmark(times = 10,
  mclapply(more_matrices, solve, mc.cores=4),
  mclapply(more_matrices, solve, mc.cores=2),
  mclapply(more_matrices, solve, mc.cores=1)
)
```

```
## Unit: milliseconds
##                                                expr      min
##  mclapply(more_matrices, solve, mc.cores = 4) 215.0294 2
##  mclapply(more_matrices, solve, mc.cores = 2) 301.9150 3
##  mclapply(more_matrices, solve, mc.cores = 1) 296.4065 3
##    median       uq      max neval
##  351.9944 455.8139 563.4863    10
##  412.5179 537.1783 595.4181    10
##  348.2005 367.4575 428.0316    10
```

There is significant overhead involved in splitting work over cores: check it is justified first!

parallel

- Argument: `mc.preschedule`
  - TRUE: short computations or large number of values in X
  - FALSE: high variance of completion time and not too many values of X compared to mc.cores
- Be careful of multiple levels of parallelisation
  - Multiple processes on multiple cores cause chaos (crashes)
  - Be careful with GUIs and parallelisation (e.g., Shiny)

parallelisation on Windows

Unfortunately it is slightly more work:

```
z <- as.list(1:4)
system.time(lapply(z, function(x) Sys.sleep(1)))
```

```
##    user  system elapsed
##   0.001   0.000   4.014
```

You need to register a cluster, then use it. Don't forget to shut it down!

```
cl <- parallel::makeCluster(4, type="SOCK")
system.time(parallel::clusterApply(cl, z, function(x) Sys.s
```

```
##    user  system elapsed
##   0.001   0.001   1.006
```

```
parallel::stopCluster(cl)
```

Writing for parallelisation

Many well-written R packages will check to see if you have `doParallel` or `doSNOW` available, and then will adapt to your system - Check to see if `ncores` (or similar) is an argument to a function that you are using - Implementing this is a little advanced; check out `DivNet` if you're interested

Writing for parallelisation: why you should

"If a program takes longer than 8 minutes for me to install, I will never ever use it, no matter how good it is."

- (Actual BFF) Chris Quince

An example of parallelisation via `parallel`

The `simulator` uses `parallel` under the hood.

```
generated_model %>%
  simulate_from_model(nsim = 40,
                      index = 5:8,
                      parallel = list(socket_names = 4)) %>
  run_method(list(lse),
  parallel = list(socket_names = 4)) %>%
  evaluate(list(squared_error))
```

Summary so far

We have now seen

- How to benchmark options
- How to run your own functions in parallel
- How some packages use `parallel` under the hood

What about large-scale data analysis?

Common in genomics, and increasingly common in modern public health

Data analysis of large data frames

```
devtools::install_github("wesm/feather/R")
```

```
## Skipping install of 'feather' from a github remote, the
##   Use `force = TRUE` to force installation
```

```
library(feather)
cakes <- feather::read_feather("iHMP_IBD.MTX.10samples.feat
cakes
```

```
## # A tibble: 62,248 x 79
##    index SAMN07424551 SAMN07424552 SAMN07424553 SAMN0742
##    <chr>        <dbl>        <dbl>        <dbl>      <
##  1 SAMN~            0            0        0.488        0
##  2 SAMN~            0            0            0        0
##  3 SAMN~            0            0            0        0
```

Preparing for data analysis

```
cakes %<>% gather(key="BioSample", "depth", -1)
cakes
```

```
## # A tibble: 4,855,344 x 3
##    index                                BioSample    depth
##    <chr>                                <chr>        <dbl>
##  1 SAMN07424251_OGJIOPKL_100510 SAMN07424551 0
##  2 SAMN07424251_OGJIOPKL_100564 SAMN07424551 0
##  3 SAMN07424251_OGJIOPKL_100587 SAMN07424551 0
##  4 SAMN07424251_OGJIOPKL_100783 SAMN07424551 0
##  5 SAMN07424251_OGJIOPKL_100920 SAMN07424551 0.535
##  6 SAMN07424251_OGJIOPKL_100933 SAMN07424551 1.45
##  7 SAMN07424251_OGJIOPKL_100967 SAMN07424551 1.66
##  8 SAMN07424251_OGJIOPKL_100995 SAMN07424551 0
##  9 SAMN07424251_OGJIOPKL_101094 SAMN07424551 0
## 10 SAMN07424251_OGJIOPKL_101127 SAMN07424551 0
## # ... with 4,855,334 more rows
```

Joining my metadata (mapping) file

```
meta <- read_csv("iHMP_IBD.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   AvgSpotLen = col_integer(),
##   MBases = col_integer(),
##   MBytes = col_integer(),
##   collection_date = col_integer(),
##   InsertSize = col_integer()
## )

## See spec(...) for full column specifications.
```

```
meta
```

```
## # A tibble: 378 x 35
##    Run   host_subject_id host_id timepoint assay_type he
```

Joining my metadata (mapping) file

```
cakes %<>% left_join(meta)
```

```
## Joining, by = "BioSample"
```

```
cakes
```

```
## # A tibble: 4,855,344 x 37
##    index BioSample depth Run   host_subject_id host_id t
##    <chr> <chr>     <dbl> <chr> <chr>           <chr>   <
##  1 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
##  2 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
##  3 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
##  4 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
##  5 SAMN~ SAMN0742~ 0.535 SRR5~ M2021C1_MTX     M2021   C
##  6 SAMN~ SAMN0742~ 1.45  SRR5~ M2021C1_MTX     M2021   C
##  7 SAMN~ SAMN0742~ 1.66  SRR5~ M2021C1_MTX     M2021   C
##  8 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
##  9 SAMN~ SAMN0742~ 0     SRR5~ M2021C1_MTX     M2021   C
```

Grab only the necessary data

```
cakes <- cakes %>%
  select(BioSample, index, depth, health_status) %>%
  filter(health_status %in% c("CD", "Non-IBD"))
```

Analyzing this data

Goal: Look for significant associations of genes (`index`) with disease state (`health_status`)

How long do we think this will take?

```
cakes %>% summarise(n_distinct(index)) %>% unlist
```

```
## n_distinct(index)
##             62248
```

```
cakes %>% summarise(n_distinct(BioSample)) %>% unlist
```

```
## n_distinct(BioSample)
##                    57
```

Microbenchmark

```
mbm <- microbenchmark(
  {y <- rnorm(78); x <- rnorm(78); lm(y ~ x)}
)
mbm
```

```
## Unit: microseconds
##                                                                  expr
##  {     y <- rnorm(78)     x <- rnorm(78)     lm(y ~ x) }
##      mean  median       uq      max neval
##  443.2479 422.173 440.8835 1588.252   100
```

```
(mbm %$% time %>% median) * 1e-6 * 62248 / 3600
```

```
## [1] 7.29984
```

Estimated: runtime of 62k regressions on 78 observations: about 8 hours

multidplyr

"`multidplyr` is a backend for `dplyr` that partitions a data frame across multiple cores. You tell `multidplyr` how to split the data up with `partition()` and then the data stays on each node until you explicitly retrieve it with `collect()`. This minimises the amount of time spent moving data around, and maximises parallel performance..."

– BFF Hadley

multidplyr

"Due to the overhead associated with communicating between the nodes, you won't expect to see much performance improvement on basic `dplyr` verbs with less than ~10 million observations..."

– BFF Hadley

Let's use multidplyr

Definitely in development but actively maintained

```
devtools::install_github("hadley/multidplyr")
```

```
## Skipping install of 'multidplyr' from a github remote, t
##   Use `force = TRUE` to force installation
```

```
library(multidplyr)
```

multidplyr

Split data up over multiple cores

```
ncores <- 4
cakes %<>%
  group_by(index)
cluster <- create_cluster(cores = ncores)
```

```
## Initialising 4 core cluster.
```

```
by_group <- cakes %>%
  partition(index, cluster = cluster)
```

```
## Warning: group_indices_.grouped_df ignores extra argumer
```

This ensures that all data in the same group goes in the same "shard"

multidplyr

```
start <- proc.time() # Start clock
processed_in_parallel <- by_group %>%
  summarise(p_val = summary(
    lm(depth ~ health_status)
  )$coef[2, 4],
  coef_est = summary(
    lm(depth ~ health_status)
  )$coef[2, 1]) %>%
  collect() %>% # function to recombine partitions
  as_tibble()
time_elapsed_parallel <- proc.time() - start
```

Always important to check

```
summary(lm(depth ~ health_status,
           data = cakes %>% filter(
             index == processed_in_parallel$index[5]
             )))$coef
```

```
##                          Estimate Std. Error   t value  Pr
## (Intercept)             1.0621427  0.8133009  1.305965 0.1
## health_statusNon-IBD -0.9563694  1.8513661 -0.516575 0.6
```

```
head(processed_in_parallel, 5)
```

```
## # A tibble: 5 x 3
##   index                          p_val coef_est
##   <chr>                          <dbl>    <dbl>
## 1 SAMN07424251_OGJIOPKL_100920 0.403   -0.311
## 2 SAMN07424251_OGJIOPKL_100995 0.589   -0.615
## 3 SAMN07424251_OGJIOPKL_101094 0.728   -0.107
## 4 SAMN07424251_OGJIOPKL_101225 0.440   -0.0964
```

Let's see how long it took

```
time_elapsed_parallel
```

```
##    user  system elapsed
##   0.047   0.029  23.653
```

This was unbelievable to me... and why I am teaching it to you!

A quick look at the data

```
processed_in_parallel  %>%
  arrange(desc(coef_est))
```

```
## # A tibble: 62,248 x 3
##    index                              p_val coef_est
##    <chr>                              <dbl>    <dbl>
##  1 SAMN07424559_ANOFLIEK_24652 0.237     1514.
##  2 SAMN07424306_PBIHEEJJ_19786 0.0783    1224.
##  3 SAMN07424567_LNNPIEOP_08899 0.0803     833.
##  4 SAMN07424568_JKFDFLGO_06732 0.159      769.
##  5 SAMN07424284_ALEGCIBH_40364 0.746      661.
##  6 SAMN07424514_CJJNBMBF_16787 0.0440     619.
##  7 SAMN07424501_DNNKMFDB_22371 0.0742     597.
##  8 SAMN07424566_MGKBKFLG_12111 0.111      569.
##  9 SAMN07424536_CCBBMNNP_23079 0.0120     456.
## 10 SAMN07424614_AOEHBOOC_04646 0.220      408.
## # ... with 62,238 more rows
```

A quick look at the data

```
processed_in_parallel %>%
  arrange(p_val)
```

```
## # A tibble: 62,248 x 3
##    index                                  p_val coef_est
##    <chr>                                  <dbl>    <dbl>
##  1 SAMN07424564_NPKIDKIG_01029  0.00000889    2.07
##  2 SAMN07424395_GBDDKALD_52613  0.0000168     0.935
##  3 SAMN07424369_ODEMOCKO_76309  0.0000214     0.768
##  4 SAMN07424520_IMOKJCLF_52792  0.0000305     0.848
##  5 SAMN07424413_AKPBDGAN_35653  0.0000362     5.62
##  6 SAMN07424369_ODEMOCKO_81434  0.0000365     0.559
##  7 SAMN07424522_HDNAFFAI_105792 0.0000450     2.55
##  8 SAMN07424334_LOEEILJP_47089  0.0000494     5.29
##  9 SAMN07424289_MODOLPOI_89206  0.000115      1.10
## 10 SAMN07424310_ONNGJNML_51693  0.000123      1.77
## # ... with 62,238 more rows
```

Save to file

```
processed_in_parallel  %>%
  arrange(p_val) %>%
  write_csv(path="cd_vs_not_50pct.csv")
```

Multiple ways to make your code run faster

- Not intelligently
- Intelligently

It's not a bad thing to have your code run faster in a not-intelligent way!

Intelligent ways of coding

In methods development (and many job interviews), you only get points for the intelligent ways

- Thoughtfully using algorithms, or developing new ones
  - There is a somewhat standard toolkit for developing statistical computing algorithms, and it varies by field
  - Think about if you want to learn more about algorithms and tell me in your feedback sheet

A common problem

```
list.files("tricky_example/")
```

```
##  [1] "file1.csv"  "file10.csv" "file11.csv" "file12.csv"
##  [6] "file14.csv" "file15.csv" "file16.csv" "file17.csv"
## [11] "file19.csv" "file2.csv"  "file20.csv" "file21.csv"
## [16] "file23.csv" "file24.csv" "file25.csv" "file26.csv"
## [21] "file28.csv" "file29.csv" "file3.csv"  "file30.csv"
## [26] "file32.csv" "file33.csv" "file34.csv" "file35.csv"
## [31] "file37.csv" "file38.csv" "file39.csv" "file4.csv"
## [36] "file5.csv"  "file6.csv"  "file7.csv"  "file8.csv"
```

Reading data from multiple files

I want to read in all of these files and append them to make one long data frame

```
read_csv("tricky_example/file1.csv")
```

```
## # A tibble: 2 x 4
##   V1    V2          V3    V4
##   <chr> <chr>  <dbl> <dbl>
## 1 D     M       0.960  1.80
## 2 T     G      -1.33  -1.34
```

```
read_csv("tricky_example/file40.csv")
```

```
## # A tibble: 4 x 4
##   V1    V2          V3     V4
##   <chr> <chr>  <dbl>  <dbl>
## 1 G     S      -0.127  0.945
## 2 U     H      -0.385 -0.139
```

Natural approach: ‘sapply

```
all_dfs <- sapply(1:40,
                  function(x) {
        read_csv(paste("tricky_example/file",
                       x, '.csv', sep=''))
                  })
all_dfs
```

```
##    [,1]         [,2]         [,3]         [,4]         [,5]
## V1 Character,2 Character,3 Character,9 Character,6 Chara
## V2 Character,2 Character,3 Character,9 Character,6 Chara
## V3 Numeric,2   Numeric,3   Numeric,9   Numeric,6   Numer
## V4 Numeric,2   Numeric,3   Numeric,9   Numeric,6   Numer
##    [,6]          [,7]         [,8]         [,9]          [,10
## V1 Character,10 Character,3 Character,8 Character,8 Char
## V2 Character,10 Character,3 Character,8 Character,8 Char
## V3 Numeric,10   Numeric,3   Numeric,8   Numeric,8   Nume
## V4 Numeric,10   Numeric,3   Numeric,8   Numeric,8   Nume
##    [,11]        [,12]        [,13]        [,14]        [,15]
```

Interesting alternative

`do.call`

- use a list to hold the arguments of the function

`sapply`

- use a vector to hold the arguments of the function

`do.call` is just like `sapply`, but better

do.call

```
all_dfs_list <- lapply(1:40,
                       function(x) {
                         read_csv(paste(
                           "tricky_example/file", x,
                           '.csv', sep=''))
                })
```

```
## Parsed with column specification:
## cols(
##   V1 = col_character(),
##   V2 = col_character(),
##   V3 = col_double(),
##   V4 = col_double()
## )
## Parsed with column specification:
## cols(
##   V1 = col_character(),
##   V2 = col_character(),
```

do.call

```
answer
```

```
## # A tibble: 224 x 4
##    V1    V2          V3     V4
##    <chr> <chr>    <dbl>  <dbl>
##  1 D     M       0.960   1.80
##  2 T     G      -1.33   -1.34
##  3 R     S       0.0900  0.921
##  4 A     Q      -1.24    0.945
##  5 X     I       0.128   0.398
##  6 N     Y       1.28   -1.03
##  7 G     R       1.08   -0.999
##  8 O     L       0.561   1.27
##  9 A     U       0.159  -0.932
## 10 E     V       0.712   1.30
## # ... with 214 more rows
```

The Max Power way

- Certain operations are expensive: avoid them
- A little thought can save a lot of time
- More on profiling and debugging soon

Coming up

- Homework 5: due next *Friday* afternoon
  - Start it after your BIOST 533 final
  - (Perk of seeing me all the time)
- Mid-quarter feedback
  - Specific comments on curriculum/syllabus welcome
  - Tell me what you want to learn and I will try to accommodate you (and everyone else)!