Computing Considerations

Ashley I Naimi

Spring 2024

Contents

1 Computing Considerations

Inevitably, at some point during your research career, you will likely run into computing constraints. There are a number of ways that these constraints can be dealt with, including the use of cloud computing (e.g., Amazon Web Services EC2 Instances), Linux computing clusters with job scheduling managers, hefty desktop computers with multiple computing cores, multiple threads, and large amounts of RAM.

However, before deploying these resources, the first line of attack is to identify bottlenecks and slow-points in the code and fix that first. Sometimes, identifying these can be a problem. In this short chapter, we'll review a few tools that can be used to optimize computing.

However, as you consider optimizing your code, consider the tradeoffs to inordinate amounts of optimization:

> We should forget about small efficiencies, say about 97

This game of profiling functions to squeeze as much speed out of a function can have two important negative consequences. First, you can spend so much time trying to speed up a function that would have finished running for its intended purpose had we not decided to embark on the path to more speed. Second, we can end up changing the program to such a degree that it is no longer easy to read and understand, and makes collaborating with others or future you nearly impossible.

These are real issues that should be considered carefully when pursuing code optimization.

2 The `microbenchmark` package

The `microbenchmark` package is a very useful tool that can often be used to answer the following question: how should I do the thing I want to do?

For example, in a previous section, we discussed the comparative speed of the `sapply` and `lapply` functions for executing iterative procedures. Let's revisit the simulation function we deployed in Section 2:

```r
expit <- function(x){
  exp(x)/(1+exp(x))
  }

set.seed(123)

simulation_function <- function(nsim, sample_size, parameter){

  # data generation
  c <- rnorm(sample_size, mean = 0, sd = 1)

  p_y <- expit(-2 + log(parameter)*c)

  y <- rbinom(sample_size, size = 1, p = p_y)

  a_data <- data.frame(c, y)

  # analysis

  mY <- mean(a_data$y)

  glm_fit <- glm(y ~ c, data = a_data, family = binomial("logit"))

  glm_res <- summary(glm_fit)$coefficients[2,1]

  sim_res <- c(nsim, sample_size, parameter, mY, glm_res)

  return(sim_res)

}
```

As a reminder, here is the output of the function under a few specific function arguments:

```
simulation_function(nsim = 1, sample_size = 500, parameter = 2)
```

```
## [1]   1.0000000 500.0000000   2.0000000   0.1180000   0.9608086
```

We can loop over this function using either `sapply` or `lapply`, and we may be interested in which one is faster. We can use the `microbenchmark` package to do this, as follows:

```
#install.packages("microbenchmark")

library(microbenchmark)

microbenchmark(
  sapply_results1 = sapply(1:100,
                           function(x) simulation_function(nsim = x,
                                                            sample_size = 500,
                                                            parameter = 2),
                           simplify = T),
  lapply_results2 = lapply(1:100,
                           function(x) simulation_function(nsim = x,
                                                            sample_size = 500,
                                                            parameter = 2)),
  times=10L
  )
```

```
## Warning in microbenchmark(sapply_results1 = sapply(1:100, function(x)
## simulation_function(nsim = x, : less accurate nanosecond times to avoid
## potential integer overflows

## Unit: milliseconds
##             expr      min       lq     mean   median       uq       max neval
##  sapply_results1 82.65292 83.56308 85.17795 84.95202 87.08613  87.75562    10
##  lapply_results2 82.90762 83.66206 91.37618 85.13982 86.43825 147.60221    10
##  cld
##    a
##    a
```

This function takes as an argument two functions that can initiate some task, as well as a `times` argument, which determines how many times each function is run. `microbenchmark` runs the function X many times (in our case, 10) and calculates how long it takes to run each time. The output represent basic descriptive statistics on the time it took to run each function, such as the median run time.

3 Introduction to Parallel Processing

One very useful technique that can save quite a bit of run time on deploying a simulation function is parallel processing. When running an R command, the instructions are sent to the CPU sequentially. For example, if we run the following `lapply` function for our simulation function defined above:

```
lapply(1:100, function(x) simulation_function(nsim = x, sample_size = 500,
    parameter = 2))
```

R will use a single computing core[1] to carry out the first iteration of the function (`nsim = 1`), and when this is complete, R will use a single computing core to carry out the second iteration of the function, and so on. However, most modern computers, including the laptop you are likely using right now, have more than a single computing core. This provides us with an opportunity to use multiple cores simultaneously to accomplish our computing goals.

[1] Note I am being deliberately ambiguous here. There are important distinctions between cores, threads, CPUs, and likely other terms that I don't quite understand the fine grain differences between. At an introductory level, these differences don't matter much. The code I will show you should work in fairly simple situations we will cover in this short course.

The basic idea behind sequential versus parallel computing can be understood via the simple diagram in Figure 1. This Figure shows how a computing job with 100 steps can be broken up into ten jobs, each with ten steps. If we can run these ten steps simultaneously, then the overall computing time consists of the time it takes to run ten steps (plus setting up the computing infrastructure needed to run in parallel), instead of the time it takes to run 100 steps.

There are a few important issues to consider before deciding to break up a job into a set of parallel runs. The first is the idea that it takes time for a given computer to set up a set of parallel workers. Depending on the circumstances, the consequence can be that running a job in parallel can take *longer* than running a job in sequence. The issue boils down to the following:

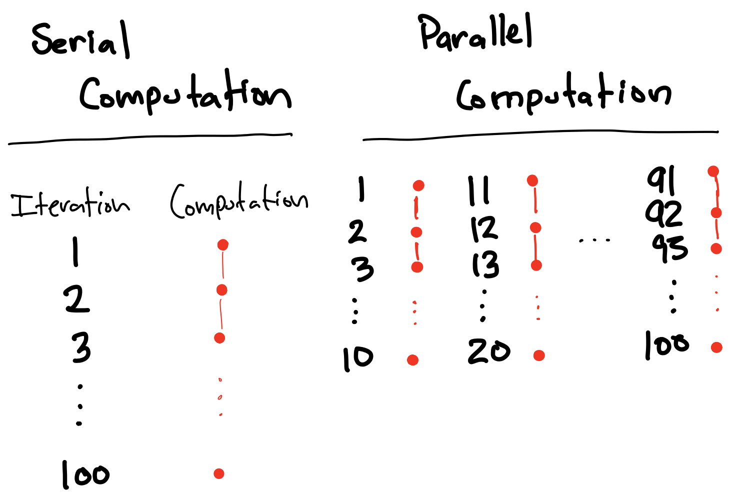

Figure 1: Simple illustration of serial versus parallel computing.

$$\text{run time}_{\text{seq}} = \text{job time}$$

$$\text{run time}_{\text{par}} = \text{job time} + \text{parallelization time}$$

In my experience, this most often occurs in simpler settings when the job itself runs quickly, but when it is parallelized into a large number of runs. In this case, it would be preferable to avoid parallel processing.

A second important concept to understand is the distinction between an *embarrassingly parallel* job, and jobs that are not embarrassingly parallel. In an embarrassingly parallel job, there is no need to transmit any information across computing cores used to parallelize the work. For example, if we refer to Figure 1, an embarrassingly parallel job is one where iterations 11 to 20 do not depend on any of the output from any of the other iterations (e.g., 1 to 10, 21 to 30, etc). If this were not the case, the job would require a degree of serialization that would complicate parallelizing. Many systems exist for this particular situation, for example, the message passing interface (MPI), which is constructed to connect different computing cores[2] where information is needed across cores to complete the desired tasks.

[2] again, likely an abuse of language here

Fortunately, in the context of Monte Carlo simulation, jobs are usually always embarrassingly parallel, and we will limit out attention to only these.

4 Parallel Processing in R

4.1 Available Computing Cores

In R, there are a number of functions available for us to deploy Monte Carlo simulations in parallel. We'll start with the `parallel` package, which combines and supercedes former packages `multicore` and `snow`. We don't need to install the `parallel` package, since it is now included as a part of base R, but we do need to load it as a standard library. Within the `parallel` package, there are several functions of use. The first is the `detectCores()` function, which tells us how many computing cores R has access to if we choose to deploy parallelization:

```r
library(parallel)

detectCores(logical = TRUE)
```

```
## [1] 12
```

```r
detectCores(logical = FALSE)
```

```
## [1] 12
```

The `logical` argument in this function enables us to count or not count the number of threads per processing core (`logical = TRUE`), versus only the number of processing cores `logical = FALSE` on a given system. Note that this function is system specific. Logical cores are not available on my laptop (MacBook Pro with Apple M2 Max Chip). However, on Windows systems with (e.g.,) Intel Core $i9$, the use of Hyperthreading creates an opportunity to split each core into threads, thus increasing the total number of cores available. In this case, the `logical` argument will return different results if `TRUE` versus `FALSE`.[3]

[3] See section 2 of the package documentation, available here: https://bit.ly/3SyS4Yu

Note that the `detectCores` function may not return accurate information on a computing cluster with workload management and system administrator limits. For example, at Emory University, I have a set of computing clusters available, with computing job parameters determined by the SLURM workload manager. Each computing node I have access to has a total of 32 cores, but the number available will depend on how many I request in the SLURM file. Unfortunately, even if I request (e.g.) ten cores in the SLURM, the `detectCores` function will return 32 (even though the limit will be ten!).

4.2 Embarrasingly Parallel Functions

In R, the simplest parallel processing functions have been built on the basis of the existing `apply` family of functions, including `lapply`. We'll cover two functions here: `parLapply` and `mclapply`. We'll start with `mclapply`, which is by far much easier to use. We'll motivate this analysis, we'll use the simulation function we constructed to explore the effect of non-collapsibility, with 50,000 simulation runs (which is very near overkill for this kind of simulation!):

```r
expit<-function(a){1/(1+exp(-a))}

set.seed(123)

collapsibility_function <- function(index, intercept){
    n=500

    C<-rnorm(n,0,1)

    theta<- c(0,log(2))
    pi <- expit(theta[1]+theta[1]*C)

    A<-rbinom(n,1,pi)

    beta<-c(intercept,log(2),log(2))
    mu<-expit(beta[1] + beta[2]*A + beta[3]*C)
    Y<-rbinom(n,1,mu)

    glm.res0<-mean(Y)

    m1<-glm(Y~A+C,family=binomial(link="logit"))
    glm.res1<-m1$coefficients[2]

    muhat1<-mean(predict(m1,newdata=data.frame(A=1,C),type="response"))
    muhat0<-mean(predict(m1,newdata=data.frame(A=0,C),type="response"))

    ## compute the odds from these average probabilities
    odds1<-muhat1/(1-muhat1)
    odds0<-muhat0/(1-muhat0)

    ## glm.res2 is the marginal log-odds ratio
    glm.res2<-log(odds1/odds0)

    res <- data.frame(intercept = intercept,
                      prevalenceY = glm.res0,
                      conditionalOR = glm.res1,
                      marginalOR = glm.res2)

    return(res)
}
```

4.3 Using `mclapply`

We can run the above function in parallel using `mclapply` as follows:

```
# choose the number of cores to use
num_cores <- detectCores() - 2

# how many cores?
num_cores
```

```
## [1] 10
```

```
# special mclapply seed ...
RNGkind("L'Ecuyer-CMRG")
```

```r
# set the seed
set.seed(123)

# run the function
sim_res <- mclapply(1:50000,
                    function(x) collapsibility_function(index = x,
                                                        intercept = -2.75),
                    mc.cores = num_cores)

sim_res0 <- do.call(rbind, sim_res)

sim_res <- mclapply(1:50000,
                    function(x) collapsibility_function(index = x,
                                                        intercept = 0), # change the intercept!
                    mc.cores = num_cores)

sim_res1 <- do.call(rbind, sim_res)

sim_res <- rbind(sim_res0,
                 sim_res1)

head(sim_res, 3)
```

```
##    intercept prevalenceY conditionalOR marginalOR
## A      -2.75       0.108     0.5044489  0.4734737
## A1     -2.75       0.102     0.9768095  0.9627058
## A2     -2.75       0.104     1.0917384  1.0464083
```

```r
tail(sim_res, 3)
```

```
##         intercept prevalenceY conditionalOR marginalOR
## A499971         0       0.558     0.3688528  0.3424711
## A499981         0       0.578     0.7753672  0.6865089
## A499991         0       0.542     0.3669095  0.3216761
```

Let’s do some basic descriptives of these results:

```
sim_res %>%
  group_by(intercept) %>%
  summarize(mY = mean(prevalenceY),
            m_cOR = mean(conditionalOR),
            m_mOR = mean(marginalOR))
```

```
## # A tibble: 2 x 4
##   intercept    mY m_cOR m_mOR
##       <dbl> <dbl> <dbl> <dbl>
## 1     -2.75 0.102 0.710 0.676
## 2      0    0.576 0.698 0.630
```

Let's unpack what we did in the code above, particularly with respect to the use of seeds here. If we were to have simply used `set.seed(123)` without anything else, we would not have been able to reproduce our results. This is because seeds work differently when using paralllel processing. However, by simply calling the `L'Ecuyer-CMRG` random number generator, which is compatible with `mclapply`, we can solve this problem.[4]

[4] Just note that the same seeds will be used every time you call the `mclapply` function. See this SO post for a trick to overcome this: https://bit.ly/3tYYRQT

4.4 Using `parLapply`

Another function we can use to deploy parallel processing is the `parLapply` function. Using this function is more involved than `mclapply`, but for some systems (i.e., Windows) it may be the only option.

```
# define the cluster type
cluster_type <- "FORK"

# number of cores
n.cores <-  detectCores() - 2

# make the cluster
cluster = makeCluster(n.cores,
                      type = cluster_type)
```

```r
clusterSetRNGStream(cl = cluster, iseed = 123)

sim_res <- parLapply(cluster,
                     1:50000,
                     function(x) collapsibility_function(index = x,
                                                         intercept = -2.75)
)

sim_res0 <- do.call(rbind, sim_res)

sim_res <- parLapply(cluster,
                     1:50000,
                     function(x) collapsibility_function(index = x,
                                                         intercept = 0)
)

sim_res1 <- do.call(rbind, sim_res)

stopCluster(cl = cluster)

sim_res <- rbind(sim_res0,
                 sim_res1)

head(sim_res, 3)
```

```
##    intercept prevalenceY conditionalOR marginalOR
## A      -2.75       0.110     0.9833969  0.9352875
## A1     -2.75       0.096     0.9764181  0.9259631
## A2     -2.75       0.098     0.9412124  0.8759729
```

```r
tail(sim_res, 3)
```

```
##         intercept prevalenceY conditionalOR marginalOR
## A499971         0       0.596     0.7992631  0.6834219
## A499981         0       0.584     0.9672604  0.8687232
## A499991          0       0.588     0.2616788  0.2361765
```

Let's again look at these results:

```
sim_res %>%
  group_by(intercept) %>%
  summarize(mY = mean(prevalenceY),
            m_cOR = mean(conditionalOR),
            m_mOR = mean(marginalOR))
```

```
## # A tibble: 2 x 4
##   intercept    mY m_cOR m_mOR
##       <dbl> <dbl> <dbl> <dbl>
## 1     -2.75 0.102 0.710 0.677
## 2      0    0.576 0.699 0.630
```

4.5 Using `parLapply` with `PSOCK`

```
cluster_type <- "PSOCK"

n.cores <- detectCores()

cluster = makeCluster(n.cores,
                      type = cluster_type)

clusterSetRNGStream(cl = cluster, iseed = 123)

clusterExport(cluster,
              c("collapsibility_function",
                "expit")
              )

sim_res <- parLapply(cluster,
                      1:50000,
                      function(x) collapsibility_function(index = x,
                                                                  intercept = 0)
)

sim_res <- do.call(rbind, sim_res)

stopCluster(cl = cluster)

head(sim_res, 3)
```

```
##    intercept prevalenceY conditionalOR marginalOR
## A          0       0.598     0.8282048  0.7592482
## A1         0       0.582     0.5494666  0.4906523
## A2         0       0.590     0.9448169  0.8619705
```

Using `parLapply` with a `PSOCK` cluster is not something I personally do. However, it may be the only option for some operating systems, such as Windows. There are two things we need to deploy `parLapply` with `PSOCK` that are different from alternative approaches. The first, we need set the seed using the `clusterSetRNGStream` function. The second is the need to export our function to each of the PSOCK clusters that is created using the `clusterExport` function. This latter one can be tricky at times. In a more complex situation, there can be several functions that we create to complete a simulation. All of them will need to be included in the `clusterExport` function.

Additionally, we may need to include libraries for specific functions that we are using (e.g., `ranger` for random forests or `xgboost` for extreme gradient boosting). Finally, in certain settings you may need to call libraries from a specific user-defined location instead of the default location. This directory will have to be included as well.

Here is a specific call to the `clusterExport` function I have used in the past:

```
parallel::clusterEvalQ(cluster, {
    .libPaths("~/R/R_LIBS_USER")
    library(SuperLearner)
    library(ranger)
```

```r
    library(xgboost)
    library(glmnet)
})
```