Parallel Computing*

Chapter 10

Anna Ziff

R Workflow for Economists

Contents

*Please contact anna.ziff@duke.edu if there are errors.

Here are the packages you will need in this chapter.

```
library(doParallel)
library(foreach)

library(AER)
library(dplyr)
```

Parallel Computing

Up to now, we have seen R code run very quickly, especially vectorized functions. However, implementing more complex functions or using bigger data can result in code that does not run or takes an unreasonably long amount of time.

Computer Vocabulary

Some knowledge of the basics of how computers work is necessary to understand parallel computing. Every computer has a Central Processing Unit (CPU), also called processors. Modern computers tend to have more than one processor. Each processor has multiple cores. You can think of each core of each processor being able to handle one computation. Figure 1 from Jones (2017) shows a stylized processor with 4 cores.

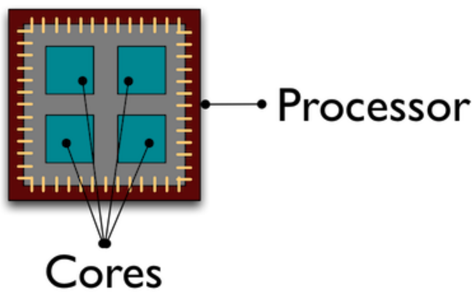

Figure 1: Stylized CPU

If you are a Mac user, type the following in Terminal to figure out how many processors and cores you have available. The first number is the number of processors. The second number is the total number of cores. My computer has 4 processors, each with 2 cores. This amounts to a total of 8 computations at one time being possible.

```
sysctl hw.physicalcpu  hw.ncpu
```

```
## hw.physicalcpu: 10
## hw.ncpu: 10
```

If you are a Windows user, open your Task Manager and select the Performance tab. This will show how many cores there are per processor. The number of "logical processors" is the number of processors times the number of cores per processor.

If you are a Linux user, type the following into your command line. The line "Core(s) per socket" will tell you the total number of cores you have.

```
lscpu
```

Usually when you run code in RStudio, it only uses one processor. Parallel computing means that you are explicitly telling R to run code in parallel across more than one processor.

When is it appropriate to parallelize?

While it may be tempting to parallelize everything, this is not the most efficient approach. When running code in parallel, the computer copies the code to each processor prepares the processor to run the code. This itself is burdensome. If the code you are running in parallel is very simple, then the advantages of parallelism will be negated by the efficiency cost.

Because each processor will run its own code, the tasks in the code need to be independent from one another. The term "embarrassingly parallel" means that it is easy to imagine your problem as several parallel tasks. These are the best candidates for parallel computing. No task depends on the result of another.

Practice Exercises 10.1

1. Recall that a grid search algorithm computes a result for all possible parameter combinations to then find the best parameters. Is this an embarrassingly parallel problem?
2. I want to calculate a function on all of the possible paths through a network. Is this an embarrassingly parallel problem?
3. I want to run my Mincer regression separately for each county in the U.S. (using different data than the built-in CPS data). Is this an embarrassingly parallel problem?
4. Looping through possible specifications, I want to change the covariates of my regression based on the outcome of the previous iteration. Is this an embarrassingly parallel problem?

Non-Parallel Implementations

Bootstrapping is a good example of an embarrassingly parallel problem. First, let us implement bootstrapping without parallelism. Here is an implementation of the bootstrap with a for loop. First, we set up the data, set the seed, define the formula, and define the number of replications. Although we can get the point estimate in the loop, we can just save this to get it out of the way.

```r
# Setup
set.seed(1024)
replications <- 100
data("CPS1985")
cps <- tibble(CPS1985)
mincer <- formula(log(wage) ~ education + experience + I(experience^2))

# Get point estimate
point <- lm(mincer, data = cps)
```

Here is the for-loop implementation. The `gcFirst` argument for `system.time()` indicates to do a garbage collection first. This allows for more comparable system times.

```r
# Bootstrap
for_results <- matrix(NA,
                  nrow = replications,
                  ncol = length(labels(terms(mincer))) + 1)

system.time(for (b in 1:replications) {

    bsamp <- cps %>%
        slice_sample(prop = 1,
                     replace = TRUE)

    for_results[b, ] <- lm(mincer, data = bsamp) %>%
        coefficients()
}, gcFirst = TRUE)
```

```
##    user  system elapsed
##   0.094   0.002   0.096
```

```
# Package results
for_results <- tibble(as.data.frame(for_results))
names(for_results) <- c("int", "edu", "exp", "exp2")
```

Before thinking about implementing parallelism, we can implement an apply-like function to see if it helps.

```
bootstrap <- function(b, form, tib) {

    bsamp <- tib %>%
        slice_sample(prop = 1,
                     replace = TRUE)

    lm(form, data = bsamp) %>%
        coefficients() %>%
        return()
}

set.seed(1024)
system.time(
    lapply_results <- lapply(1:replications, bootstrap, form = mincer, tib = cps),
    gcFirst = TRUE)
```

```
##    user  system elapsed
##   0.088   0.002   0.090
```

Parallel Implementations

There are several different packages and functions available for parallel implementation in R.

`mclapply`

The function `mclapply()` comes from the `parallel` package. The syntax and output is the same as `lapply`. There is a small detail that setting the seed is less straight-forward in parallel functions. A simple solution for `mclapply()` is to define the random number generator and then allow `mclapply()` to set the seed. Although this will not be identical to the above results, it will at least be reproducible. More information on this issue is available here.

```
RNGkind("L'Ecuyer-CMRG")
system.time(
    mclapply_results <- mclapply(1:replications, bootstrap, form = mincer, tib = cps,
                                 mc.set.seed = TRUE),
    gcFirst = TRUE)
```

```
##    user  system elapsed
##   0.001   0.003   0.073
```

It is possible to explicitly set the number of cores.

```
# Double check the number of cores
n_cores <- detectCores()

# Max out number of cores
system.time(
    mclapply_results <- mclapply(1:replications, bootstrap, form = mincer, tib = cps,
                                 mc.set.seed = TRUE,
                                mc.cores = n_cores),
    gcFirst = TRUE)
```

```
##    user  system elapsed 
##   0.099   0.095   0.035
```

If you are going to be using `mclapply()` in your code, it is suggested NOT to use RStudio. This function may not be available for Windows operating systems. You can see if `parLapply()` will work instead.

`foreach`

The packages `foreach` and `doParallel` allow for a parallel implementation of a for loop. The syntax is just a little different, and can actually be used for non-parallel implementation.

```
# These are the same, but the output is a little different
for (i in 1:3) {
    print(factorial(i))
}
```

```
## [1] 1
## [1] 2
## [1] 6
```

```
foreach (i = 1:3) %do% {
    factorial(i)
}
```

```
## [[1]]
## [1] 1
## 
## [[2]]
## [1] 2
## 
## [[3]]
## [1] 6
```

To implement parllelism with `foreach`, it is necessary to setup the number of cores. The `doParallel` package allows for this.

```
# Start the cluster
registerDoParallel(n_cores)

foreach (i = 1:3) %dopar% {
    factorial(i)
}
```

```
## [[1]]
## [1] 1
## 
## [[2]]
## [1] 2
## 
## [[3]]
## [1] 6
```

```
# Stop the cluster
stopImplicitCluster()
```

Note that the default output of `foreach` is a list. This makes it convenient to save the output without initializing an object to save results.

```r
# Start the cluster
registerDoParallel(n_cores)

out <- foreach (i = 1:3) %dopar% {
    factorial(i)
}

out
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 6
```

If you do not want a list, change the `.combine` argument.

```r
# Output a vector
foreach (i = 1:10, .combine = c) %dopar% {
    factorial(i)
}
```

```
##  [1]       1       2       6      24     120     720    5040   40320  362880
## [10] 3628800
```

```r
# Output a matrix
foreach (i = 1:10, .combine = rbind) %dopar% {
    factorial(i)
}
```

```
##              [,1]
## result.1        1
## result.2        2
## result.3        6
## result.4       24
## result.5      120
## result.6      720
## result.7     5040
## result.8    40320
## result.9   362880
## result.10 3628800
```

```r
foreach (i = 1:10, .combine = cbind) %dopar% {
    factorial(i)
}
```

```
##      result.1 result.2 result.3 result.4 result.5 result.6 result.7 result.8
## [1,]        1        2        6       24      120      720     5040    40320
##      result.9 result.10
## [1,]   362880   3628800
```

```
foreach (i = 1:10, .combine = cbind) %dopar% {
    c(i, factorial(i))
}
```

```
##      result.1 result.2 result.3 result.4 result.5 result.6 result.7 result.8
## [1,]        1        2        3        4        5        6        7        8
## [2,]        1        2        6       24      120      720     5040    40320
##      result.9 result.10
## [1,]        9        10
## [2,]   362880   3628800
```

Here is the `foreach` parallel implementation of our bootstrapping problem. Again note that setting the seed is a little more complicated.

```
registerDoParallel(n_cores)
set.seed(1024, kind = "L'Ecuyer-CMRG")

system.time(foreach_results <- foreach (b = 1:replications, .combine = cbind) %dopar% {

    bsamp <- cps %>%
        slice_sample(prop = 1,
                     replace = TRUE)

    lm(mincer, data = bsamp) %>%
        coefficients()
}, gcFirst = TRUE)
```

```
##    user  system elapsed
##   0.142   0.141   0.048
```

```
stopImplicitCluster()
```

Practice Exercises 10.2

1. Define a scalar to represent data collected on the number of heads from 100 coin flips with the below code. The likelihood with parameter `p` can be calculated directly with `dbinom`. Write a `foreach` loop in parallel that calculates the likelihood for a sequence of possible parameters between 0.2 and 0.7 spaced by 0.005. Outside of the loop, find the parameter value at which the likelihood is maximized.

```
# Create data
heads <- rbinom(1, 100, 0.5)

# Calculate likelihood of parameter p
dbinom(heads, 100, p)
```

Further Reading

See Roger D. Peng's book for more background on parallel computing in R.

References

Jones, Matt. 2017. "Quick Intro to Parallel Computing in R." https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html.