Week 14: Parallel Computing

Ellen Bledsoe

2024-04-25

Parallel Computing

Introduction

In our final week of class this semester, we are going to talk about how you can utilize the power that your computer (likely) has to your benefit when performing computationally complex tasks.

Sometime when we code, we run into some issues:

- process taking too long? You might not have enough CPU (central processing unit) power
- data too large? You might not have enough memory

We can get around some of these issues by telling R to utilize more of our computer than it does by default. In this lesson, we will mostly be dealing with addressing the first issue of a process that is taking a long time to run, but we will briefly mention the memory issue later.

To understand what we are about to do, let's take a look at this diagram below. We are most interested in the processor and the core.

The processor below is a CPU (central processing unit) that retrieves data and runs instructions. Most modern computers have more than one CPU these days.

Within the CPU, we can have cores. Each core is capable of retrieving data and performing tasks.

By default, R is actually only running on a single core in a single CPU. But many computers have more than that! My laptop, for example, has 16 cores that I can access.

If we have (relatively) independent tasks that need to be accomplished, we can image a scenario where we can ask multiple cores to perform a different iteration of the same task at the same time rather than waiting for one core to perform the same task over and over again.

This lesson will demonstrate 2 different ways to harness the power of our computers more intentionally when the need arises:

1. through the `mclapply` function in the `parallel` package
2. through combined use of `foreach` from the `foreach` package and the `%dopar%` operator in the `doParallel` package

Setup

We will be using a handful of new packages while learning about parallelization in R, and it is good practice to install and load those packages at the top of the script, so let's go ahead and do so.

The `parallel` package is already installed with R, so we do not need to install it.

We will also be using the `tidyverse` and `portalr`, so install those onto your computer as well, if you haven't already.

```r
# install.packages("doParallel")
# install.packages("foreach")
# install.packages("portalr")
```

And we load them into our working environment with the `library()` command.

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

```
## Loading required package: parallel
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x purrr::accumulate() masks foreach::accumulate()
## x dplyr::filter()     masks stats::filter()
## x dplyr::lag()        masks stats::lag()
## x purrr::when()       masks foreach::when()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

Sourcing Files

In order to demonstrate a real example where parallelization might come in handy, I have modified one of the functions I wrote for a paper a few years ago. The function is stores in the `capt_hist_fxn.R` script.

We want to be able to use that function in this .Rmd file. We can tell R to read that R script so we have access to the function.

We do this with the `source()` function, which will run the entire script and put any outputs into our environment.

```r
source("capt_hist_fxn.R")
```

Creating Capture Histories

We are going to create capture histories for each individual rodent caught at the Portal Project. Capture histories indicate whether the individual was observed (caught) during a given survey occasion.

Let's pull in our data to work with.

Explicitly Calling Functions

You might also notice that I'm referencing the package name, the use `::` and then give the name of the function.

Because this lesson is using functions from a lot of different packages, I want you to be clear on which functions come from which package, so I am explicitly calling each function.

When you do this, you can use functions even if you haven't brought them into your workspace with the library function, as long as they are installed.

```r
ratdat <- portalr::summarize_individual_rodents()
```

```
## Loading in data version 5.63.0
```

```r
ratdat <- ratdat %>%
  filter(tag != "0", !is.na(tag))
```

Because this is a large dataframe, let's create a smaller version for now with only the first 1000 rows.

```r
ratdat_small <- ratdat[1:1000, ]
```

The `create_capture_hist()` function that we have read in has 2 arguments.

1. The first is `tag`, which is the tag given to an individual rodent to let us know if we have caught that individual rodent before or not.
2. The second is a dataset that includes all individual capture events

Let's first create a vector with all of the unique tags in our `ratdat_small` dataframe. Then we can randomly choose 1 tag to see how to function output looks.

```r
# requires a dataframe of individual captures and a single tag
tags <- unique(ratdat_small$tag)

# and choose a tag to test the function
test_tag <- tags[123]
```

Now, let's test the function.

```r
create_capture_hist(tag = test_tag, data = ratdat_small)
```

```
##   capture_history  tag
## 1 000010111011010 4668
```

It looks like our output is a combination of 2 different values: the capture history and then the tag. If we run a `for` loop over the `tags` vector with the function, we can create a dataframe with 2 columns.

We will use the `bind_rows` function from the `tidyverse` to add each new row to the dataframe.

```r
# create empty data frame
capt_histories <- data.frame()

# for loop iterating over the tags vector
for (i in 1:length(tags)) {
  # function output
  output <- create_capture_hist(tags[i], ratdat_small)
  # bind output from an iteration to the dataframe
  capt_histories <- bind_rows(capt_histories, output)
}

head(capt_histories)
```

```
##   capture_history  tag
## 1 100000000000000 4899
## 2 100000000000000 4825
## 3 100100000110000 4829
## 4 100101000000000 4891
## 5 111110000000000 4826
## 6 111000000000000 4819
```

The for loop worked pretty well!

But that was only with 1000 rows (and 343 tags), and we have over 63,000 rows to iterate through. What if we try 10,000 rows?

```r
ratdat_med <- ratdat[1:10000, ]
tags <- unique(ratdat_med$tag)

# create empty data frame
capt_histories <- data.frame()

for (i in 1:length(tags)) {
  output <- create_capture_hist(tags[i], ratdat_med)
  capt_histories <- bind_rows(capt_histories, output) #rbind also works
}
```

It is starting to take a bit longer to run this code…

What are other ways we could run the same thing?

The `apply` Family

One thing that people suggest is the `apply` suite of functions. Functions in the `apply` family are from base R. They allow us to iterate over many types of data structures (data frames, lists, etc.) and perform the same task on each section. People prefer them to `for` loops because they can be significantly faster when datasets get large.

There are a number of functions in the `apply` family. Some examples:

- `apply` iterates over a data structure with rows and columns
- `vapply` iterates over a vector
- `lapply` iterates over a list

Lists are by far the most flexible data structure in R. They can be seen as a collection of elements without any restriction on the class, length or structure of each element.

We aren't going to delve into these functions too much this lesson, but we will practice with `lapply` specific to serve our purposes of expanding to parallel computing.

The `apply` functions vary a bit in structure, but the `lapply` function has the following structure:

- data object to iterate over
- name of the function to apply to the data object
- any arguments (in addition to the data object) that need to be passed to the function

```r
results <- lapply(tags, create_capture_hist, data = ratdat_med)
```

To convert the results into a dataframe, we can use the `do.call` function to treat each list as a row and bind them together.

```r
# covert the results list to a data frame
results <- do.call(rbind, results)
```

This is still taking a while to run. Why don't we harness more of the power of our computer to help this run?

Parallelization

There are a number of ways that we can leverage the power of parallelization in R. We are going to cover 2 of the many ways in this lesson.

1. `mclapply` from the `parallel` package

The `mclapply` function allows us to use the `lapply` function while using more than 1 core.

First, we need to figure out how many cores our computer has. We use the `detectCores()` function to do this. It is good practice to divide by 2 or subtract 1 so you don't take over your whole computer with R!

```r
nCores <- detectCores() / 2
```

We can now use the `mclapply` function just as we did with `lapply` but with the added `mc.cores` argument.

```r
results <- mclapply(tags, create_capture_hist, data = ratdat_medium, mc.cores = nCores)
results <- do.call(rbind, results)
```

Using `parLapply`: System Agnostic! As we saw in class on Tuesday, `mclapply` didn't work on a Windows machine the way we were expecting it to.

There are some complicated reasons for this, but let's leave it at the fact that Windows machines are build differently than Mac (and Linux) machines, which makes parallel computing on them a little more difficult in the backaground.

Fortunately for us, there are some work-arounds that we can actually use on *any* system.

Instead of `mclapply`, we can use the system-agnostic `parLapply`, which is actually also part of the `parallel` package.

To use `parLapply`, we follow a very similar logic as `lapply` and `mclapply`, but we need to "make a cluster" first. This cluster then becomes the first argument in the `parLapply` function. Everything else is basically the same!

```r
# group our cores together
clust <- makeCluster(nCores)

# specify our cluster as the first argument
results <- parLapply(clust, tags, create_capture_hist, data = ratdat_med)
```

You might notice that `parLapply` ran faster than `mclapply`, so I'll probably stick to teaching the `parLapply` method in the future!

2. The `foreach` and `doParallel` Packages

Not everyone loves the `apply` family structure (…such as me), so let's introduce an alternative. I've confirmed that this route is also system agnostic.

Before we can utilize the parallel computing power of the `foreach` package and it's modifiers, we need to learn how to structure a `foreach` function.

The `foreach` function is basically a `for` loop that has been turned into a function. Some differences to note:

- we want to save the output as an object
- we set `i = 1:length` to a length to iterate rather than `i in 1:length`
- we can specify the format of the results with the `.combine` function
- we use the `%do%` operator

Let's try it out!

In this case, because the function's output is essentially 1 row, we are going to tell `foreach` to use the `rbind` function (base R version of `bind_rows`) to bring the results together.

```r
output <- foreach(i = 1:length(tags), .combine = rbind) %do% {
  create_capture_hist(tags[i], ratdat_med)
}
```

Parallelization with `foreach` We can modify the `foreach` function that we have written above to run on multiple cores by initiating the number of cores and changing the `%do%` operator to the `%dopar%` operator.

We have to tell the `doParallel` function to start using multiple cores with the `registerDoParallel` function. We then stop it with `stopImplicitCluster`.

```r
# set number of cores
# nCores <- detectCores() / 2

doParallel::registerDoParallel(nCores)

output <- foreach(i = 1:length(tags), .combine = rbind) %dopar% {
  create_capture_hist(tags[i], ratdat_med)
}

doParallel::stopImplicitCluster()
```

We can use the `foreach` function and `%dopar%` operator to run our function for the entire dataset.

```
tags <- unique(ratdat$tag)

doParallel::registerDoParallel(nCores)

output <- foreach(i = 1:length(tags), .combine = rbind) %dopar% {
  create_capture_hist(tags[i], ratdat)
}

doParallel::stopImplicitCluster()
```

High Performance Computing at UA

If you are working with data that would benefit from parallelization (and especially parallelization on a large scale), consider learning about the High Performance Computing that UA offers.

HPC access is available to anyone affiliated with UA, with a certain number of computing hours offered for free each month. You will likely need to be linked to a research group (AKA your PI's account).

Starting to work with the HPC system can be daunting, but UA has some really nice documentation available.

They also have an RStudio interface to access!

Resources

Parallel computing is still something I am learning myself! This lesson was heavily informed by the following sources which you might also find helpful:

- Beyond Single-Core R slidedeck
- Quick Intro to Parallel Computing in R by Matt Jones at NCEAS
- Parallelizing Code in R blog from the Weecology Lab
- `doParallel` vignette
- Software Carpentry lesson on parallel computing