Easy parallelization in R (using the foreach package)

Dan Turner (turnerdan dot com)

For a live (Zoom) workshop on March 14th, 2022.

Welcome to this R workshop about loops loops loops!

Before we can begin, please make sure that you can run the code chunk below.

Code chunks are areas of "R Markdown Notebooks" that allow us to execute our scripts. You can use the "Run" commands at the top of the code editor pane to run all or parts of chunks, or use the Cmd + Enter shortcut to Run any highlighted code, or the current line.

```r
# Set the root to point to wherever /howtoforeach/ is on your computer
setwd( "~/Git/howtoforeach" )  # Change

## Load required packages

# For workshop related tasks
library( tictoc )     # for obtaining accurate code run time
require( lme4 )       # we will bootstrap a few models; only using a few functions so saving memory by
library( beepr )      # sometimes I like an auditory notification when long processes finish

# For making foreach run correctly
library( tidyverse )  # for data processing and better syntax
library( foreach )    # for parallel loop procedures across cores
library( parallel )   # for setting up your machine for multi-core processing
library( doParallel ) # for setting up your machine for multi-core processing (also)
```

We will do some additional setup specific to `foreach` when that package is introduced.

Part 1: Getting started

To loop or not to loop

Beginning coders often start by writing looping scripts because they highlight the power of using computers to automate tasks. In general, "for" loops are used for tasks with a clear end point, and "while" loops are used for tasks with indefinite ends points.

For example, if you want to repeat a sequence for every row in a dataframe, you would use `for()`, but if you want to repeat a sequence until a condition is met, you would use a `while()` loop, which runs while the argument of `while()` remains `TRUE`.

You may have received advice to "not write loops", but there is nothing inherently incorrect about writing loops; it's legitimate code! This advice comes from the fact that some people believe that slow code is bad code, but I disagree.

Sometimes, writing a loop is easier because it follows a sequential flow that humans are used to reasoning through. When R starts breaking up calculations across cores, rewriting code into faster languages, and other performance-enhancing strategies, sometimes it's difficult to understand what is really happening at the same level.

Also, sequential processing is inherent to many computational tasks, such as text processing, repeated sampling (bootstrapping), and web scraping. So, rather than preaching that nobody should write another loop, I will show you how to write loops that are more computationally efficient.

What's the solution?

Let's assume we have to write our code as a loop for some reason, but we also want the code to execute as quickly as possible. The main reason why loops tend to run slow is due to their sequential nature–the computer has to wait to process Item 100 until it can process Item 99–and every processing delay piles up.

Our solution will be to split up the single sequence into multiple chunks, so Items 1 through 33, 34 through 67, and 68 through 100 will be processed in parallel, cutting execution time up to 66%.

"I need the absolute fastest code!"

I highly recommend the `tidyverse` collection of R packages. If you can substitute a loop for a tidyverse statement, it will be orders of magnitude faster in most cases. It is also faster to write, and easier to understand, which can be just as important as having a working script.

If you don't mind learning different syntax, you can also look at `data.table` which is generally faster than `tidyverse`, but is limited to rectangular (column-based) data.

"Vectorized" is a catch-all term for these super-fast packages that operate at multiple points of the data at the same time. Sometimes vectorized code is also run in parallel–it depends on the package and what the specific process is.

Part 2: The humble loop

Loading data files with a loop

In the next code chunk, you'll see the humble loop loading our text data files, and a vectorized version for comparison. We will time our operations using the package 'tictoc'.

Notice in this example how much more work goes into writing a loop, but it's clear and easy to modify. For example, I can write a progress message into the loop, but not into its vectorized equivalent.

Also, see how I have to create a container for the looped data; otherwise each loop would overwrite the previous one, and we would not be able to access the results of each iteration.

```r
################################
# Sequential (looping) import #
################################

# List the files in the /data/ directory to build the filepaths
filepaths = paste0( getwd(), "/data/", list.files( path = "./data/", pattern = ".csv"))

# Let's take a peek at the first file, file.1
file.1 = read_csv( filepaths[[1]], col_types = cols())  # TIP: col_types = cols() suppresses warnings

# Empty version of file.1 (so I don't need to specify colnames later)
reviews.seq = file.1[FALSE,]

# Looping the list of files (slow and lots of lines)
tic("Looping version")  # start timer
  # I am demonstrating a 'for' loop for now, but there are also 'while' loops
  for( file in seq_along( filepaths ) ) {
    print( paste("Loading file", file, "out of", paste0(length(filepaths)) ) )  # try this with vectori
    the.file = read_csv( filepaths[[ file ]], col_types = cols() )  # read the data into the.file
    reviews.seq = rbind( reviews.seq, the.file)  # add it the.file to the bottom of reviews.seq
  }
```

```
## [1] "Loading file 1 out of 3"
## [1] "Loading file 2 out of 3"
## [1] "Loading file 3 out of 3"
```

```r
toc()  # end timer
```

```
## Looping version: 0.335 sec elapsed
```

For comparison, all of the above is accomplished in 1 line here:

```r
#####################
# Vectorized import #
#####################

tic("Vectorized version")  # start timer

# With the plyr package, we can apply read_csv directly to the filepaths list
reviews = plyr::ldply( .data = filepaths, .fun = read_csv, col_types = cols() )

toc()  # end coder timer
```

```
## Vectorized version: 0.237 sec elapsed
```

```r
#  Hopefully the two objects are of the same size.
ifelse( identical( dim(reviews), dim(reviews.seq) ),
        "PASS: Results match in size", "FAIL: Results do not match in size")
```

```
## [1] "PASS: Results match in size"
```

We will conclude our comparisons of looping versus vectorized code here… if you want speed, avoid sequential operations and instead use functions that automatically vectorize the computation, such as `tidyverse` or `data.table`.

Choosing a computational environment

Before we transition into executing parallel code, let's talk about another important factor in performance–how much processing power is available? If you need more capacity than a single computer can provide, then you may want to run the code on a purpose-made computational environment.

For example, Northwestern University maintains a research computing cluster called Quest with 11,800 cores, which are optimized for running code for its students and faculty. Basic Access is free. A non-free option is Amazon AWS, which will allow you to (glossing over details) create a virtual computer for you to run code on.

To state the obvious, if your code will be running for a few days, it's more convenient to have it run on a non-essential non-personal computer!

More info: https://www.it.northwestern.edu/research/user-services/quest/overview.html

---

Part 3: The parallel loop

Text processing example

Text is typically slow to process, for a variety of reasons, and the way we work with the data during text analysis is often sequential. This makes text processing a good candidate for use of parallel loops.

For this example, we will use the data we loaded, which consists of ~50k job reviews from Amazon employees, scraped from indeed.com. Expect this chunk to take extra time to process.

```r
# For each row, let's extract the state and count the length of the review details
{ # start timing
tic("Sequential text processing")

for ( row in 1:nrow( reviews ) ) {

  # Using a regular expression to extract pairs of capitalized letters
  reviews$state[row] = str_extract( reviews$Work_Location[row],"\\b[A-Z]{2}")

  # nchar() counts the characters
  reviews$nchar[row] = nchar( reviews$Review_Details[row] )

}

toc() } # end timing
```

```
## Sequential text processing: 30.693 sec elapsed
```

Now let's split the loop over our cores, using foreach

First we have to set up the computer to run our loops in parallel. Out of the box, `foreach` does not run in parallel; it requires the `doParallel` package in order to use multiple cores. This only has to be done once per R session, and usually I do this immediately after loading libraries.

```
# Returns the number of cores that R can see
n.cores = detectCores()

# I like to keep one core in reserve. Otherwise, if the R session crashes, it can make your whole compu
cl = makeCluster( n.cores - 1 )

# Initiate an R process on the earmarked cores
registerDoParallel( cl )
```

Now that we have N-1 cores ready to receive commands from R, we will take the previous loop and divide it across the available cores. The following chunk calls `foreach()` to create a loop of row indexes which we will use to process each review in two ways: 1. We will attempt to extract the state abbreviation code (e.g. IL, CA, NY) from the review. 2. We will count the number of characters in the review. The results will be passed to `rbind` to create an extended version of the original data.

```
{  # begin timing
tic("Parallel text processing")

# Same functions as our sequential loop, but using foreach() and %dopar%
reviews.processed = foreach( row = 1:nrow( reviews ),
         .combine = "rbind",
         .packages = c("stringr")) %dopar% {  # Packages need to be loaded for each core

          # Extract state code (2 capital letters)
          the.state = str_extract( reviews$Work_Location[row],"\\b[A-Z]{2}")

          # Extract review length
          the.nchar = nchar( reviews$Review_Details[row] )

          # I'm essentially appending 2 columns to the original data by extending each row
          return( c( reviews[row,], the.state, the.nchar ) )

}  # end of foreach loop

toc() } # end timing chunk
```

```
## Parallel text processing: 36.929 sec elapsed
```

```
# Adding pretty column names to our extended data
reviews.processed = as.data.frame( reviews.processed )
colnames( reviews.processed ) = c(colnames( reviews ), # starting point
                             "state.code", # I could also left join
                             "review.len") # but this is fast and clean

reviews = reviews.processed
```

Benchmarks

Dan's 2012 MacBook (no Zoom): 45s

Data chunking example

In the previous example, we looped each row by creating an iterator out of `reviews`. This is not a great use of parallel processing though, because there is little to gain when splitting 48k rows across a few cores. It's usually more efficient to break up your data into bigger chunks, for example, by state. You always want to parallelize the slowest part of the process, if possible.

For example, I usually use foreach() when web scraping because the slowest part of the process is making a connection with the remote server, so I have each core processing a different URL usually. I make sure I only ever have to call a URL once, and once I have made a connection, I used vectorized code to extract the target data, which minimizes the time per page.

```
# Let's get a list of all the (confirmed) states that appear in the $state field. We confirm states by
state.list = as.data.frame( table( unlist( reviews$state ) ) )
the.states = intersect( state.abb, reviews$state )

{  # beginning of timer
tic("Parallel loop of all states")

# Parallel loop of the states
reviews.bystate = foreach( state = seq_along( the.states ),
         .combine = "rbind",
         .packages = c("tictoc")) %dopar% {  # let's do it in base R to avoid calling packages here

           # Collect the rows for this state
           the.state = unlist( the.states[state] )
           state.rows = reviews[ which( reviews$state.code == the.state ), ]

           # Let's time each state, out of curiosity
           tic(paste("State:", the.state))

           # Sum the nchar column
           state.sum = sum ( as.integer( state.rows$review.len ) )

           # Let's get the average rating too, while we're at it
           state.avg = round( mean( as.integer( state.rows$Rating ) ) )

           # Returning a vector on each loop, which I rbind into a matrix (fast!)
           return( c(the.state, state.sum, state.avg, toc()$toc) )

}  # end reviews.bystate
toc() }  # end of timer
```

```
## Parallel loop of all states: 2.335 sec elapsed
```

```r
# Turn the matrix into a df w/ pretty colnames
reviews.bystate = as.data.frame( reviews.bystate )
colnames( reviews.bystate ) = c( "state.code", "review.len.sum", "rating.mean", "processing time")
```

It's much faster to process state-by-state versus row-by-row, but why?

Why is it faster to loop state-by-state?

The 47 states represented in the data will be equally divided across multiple cores on my computer. For example, since I am allowing R to utilize 3 cores, each processor only has to handle about 15 states, rather than a single core handling all 47 one at a time.

Maybe Core #1 processed Michigan and Florida and Core #2 processed Illinois and California. We have a lot of control over this process, but by default foreach() is optimized to balance performance with being a general purpose tool.

Using foreach() is a little bit like dividing your data into thirds and having two colleagues process them for you while you run your share. At the end of the day, you'll have 3 sets of results which, when combined, are identical to if you ran the process sequentially.

Coding Challenge A

The first challenge involves repairing the `$state` variable, which you can tell by looking at `state.list` contains many errors. Here's some meta-code that shows the basic process I will guide you through:

Split Place_Of_Work_1 to get state

Loop city,state strings to extract state abbreviation `foreach(city_state_string)`

Validate extracted data using built-in list, state.abb `if(state in state.abb) str_extract(state)`

```r
# write_rds( reviews, "./data/snapshot_a.rds")  in coding_challenges you will load this file
```

Part 4: Going deeper into `foreach()`

Now that we've covered the basic setup and syntax of `foreach()`, we will talk more about its parameters and use cases.

Combining results

Besides being able to use multiple iterators to generate loops, `foreach` is also flexible in how it formats its output. Depending on how much memory a

Concatenate `.combine = 'c'`

I use this method when I want the result to be a list, or list of lists.

Column/row bind `.combine = 'cbind'`/`.combine = 'rbind'`

I use this method when I want the result to be a dataframe, tibble, or matrix.

Basic operators `.combine = '+'`/`.combine = '*'`

These methods make sense for bootstrapping and simulating data.

Custom function `.combine = 'cfun'`

```
# Adapted from the foreach documentation
comb.c     = foreach(i=icount(4), .combine='c') %dopar% exp(i)
comb.cbind = foreach(icount(4), .combine='cbind') %dopar% rnorm(4)
comb.add   = foreach(icount(4), .combine='+') %dopar% rnorm(4)

# TIP: You can include conditional logic to foreach calls using `%:%` and `when()` ... somewhat like Py

# As long as `i` is between 1-5, wrote the corresponding letter to the console
foreach( i = icount( 32 ) ) %:% when( i <= 5 ) %dopar% letters[i]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
##
## [[5]]
## [1] "e"
```

Nested `foreach` loops

A common text analysis problem is computing the similarity score between different strings. There are a lot of scores we could use, but to keep things simple, we simply take the proportion of words in common.

Loops can contain other loops, which can be convenient for checking all combinations of variables. In our toy problem, we are comparing vectors of words, so the first thing we need to do is create a dataset consisting of these vectors.

```
# Create a list of vectors, one for each review, containing a vector of words
grid_words = unlist( reviews$Place_Of_Work_1, reviews$Place_Of_Work_2 ) %>%
  tolower() %>%
  str_extract_all( pattern = '[a-z]+' )

# Print the head below
head( grid_words )
```

```
## [[1]]
## [1] "amazon"   "shopper"  "current"  "employee" "dedham"   "ma"       "december"
```

```
##
## [[2]]
## [1] "warehouse" "team"      "member"    "former"    "employee"  "richmond"
## [7] "va"        "december"
##
## [[3]]
##  [1] "warehouse"  "order"      "picker"     "driver"     "both"
##  [6] "inbound"    "and"        "outbound"   "capacities" "former"
## [11] "employee"   "charleston" "tn"         "december"
##
## [[4]]
##  [1] "sortation" "associate" "at"        "delivery"  "station"   "former"
##  [7] "employee"  "austin"    "tx"        "december"
##
## [[5]]
## [1] "warehouse"   "worker"      "former"      "employee"    "pendergrass"
## [6] "ga"          "december"
##
## [[6]]
## [1] "amazon"    "warehouse" "associate" "current"   "employee"  "brooklyn"
## [7] "park"      "mn"        "december"
```

The easiest way to exhaustively compare two vectors is by using `expand.grid()` which finds all combinations. For example:

```
dim( expand.grid( state.abb, state.abb ) )
```

```
## [1] 2500    2
```

But this will not work for grid_words, because it's too large

```
toobig = try( expand_grid( grid_words, grid_words ),
          silent = TRUE )

dim( toobig )
```

```
## NULL
```

This seems to be an example of an operation better accomplished through loops, since it's computationally intensive and we don't have the resources to run a vectorized version of this script.

Since loops break up the computation, it's possible to run long processes (with interruptions, although I'm not showing that).

```
# For the purposes of this workshop, you may want to down-sample before this operation (start small)
loop_words = sample( grid_words, 100 )

# Parallel loop of loops to compare
{tic("Exhaustive similarity comparisons")  # timing starts
reviews.compared =
  foreach( loop.outer = seq_along( loop_words ), .combine='rbind' ) %:%
    foreach( loop.inner = seq_along( loop_words ), .combine='c', .packages = 'dplyr' ) %do% {
      # Count words total and words in common, using set theory (fast)
      words.inner     = length( loop_words[ loop.inner ][[1]] )
      words.total     = length( union( loop_words[ loop.outer ][[1]], loop_words[ loop.inner ][[1]] ))
      words.in.common = length( intersect( loop_words[ loop.outer ][[1]], loop_words[ loop.inner ][[1]]

      # Calculate the.score
      # the.score = words.in.common / words.total words.inner
      the.score = words.in.common / words.inner

      # Return the output
      return( the.score )
    }
toc()}  # timing ends
```

```
## Exhaustive similarity comparisons: 3.294 sec elapsed
```

```
# DING
# beep()
```

Benchmarks

10…..0.1s 100…3.3s 500…73.7s 1k…321s 2k…1252s

Coding Challenge B

The second challenge involves finding the mean `$Rating` of each city/state combination. Create an outer loop that lists the cities for which there are reviews, then create an inner loop that calculates the mean `$Rating` on each iteration. Here's some meta-code to get you started thinking about this problem:

Outer Loop `foreach(state) %:% city_list = unique cities in state`

Inner Loop `foreach(city) %dorpar% calculate the mean review$Rating`

Good luck!

```
write_rds( reviews, "./data/snapshot_b.rds")  # in coding_challenges you will load this file
```

Bootstrapping

So far the examples have focused on `for` loops iterating over lists and dataframes, but this is not the whole story. For many purposes, you might want to use a `while` loop, which terminates when/while certain conditions are being met.

Another common use for loops is bootstrapping, which involves running a process for a specified number of times. For example, we can run the same model 1,000 times over different subsets of the same data and compare the results…

```r
# Let's time our bootstrap loop
{
tic("Bootstrapping a linear model n times")

bootstrap = foreach( icount( 1000 ),  # how many loops?
                     .combine  = rbind,  # each run of the model will generate 1 row
                     .packages = c ("dplyr", "lme4")) %dopar% {  # parallel mode
                     #.packages = c ("dplyr", "lme4")) %do% {  # non-parallel mode

    # Sample some random rows to make things run faster
    sample = reviews %>% sample_n(nrow(reviews) * 0.33, replace = FALSE)

    # Fit the model predicting sentiment based on length and rating
    model = lm( data    = sample,
                formula = unlist(Rating) ~ unlist(nchar) )

    # Extract model coefficients
    coefs = coefficients( model )

    # Output
    return( coefs )
  }

toc()
}
```

```
## Bootstrapping a linear model n times: 37.532 sec elapsed
```

```r
# We can get the mean of each coefficient as follows
colMeans( bootstrap )
```

```
##   (Intercept) unlist(nchar)
##   3.478518631  -0.002867332
```

The parallel version typically runs 33% faster for me.

Coding Challenge C

For this challenge, you will be bootstrapping a model of review sentiment, based on a quick text analysis that results in each review being assigned a score based on how positive or negative, according to an unsupervised algorithm. We will be trying to predict the sentiment based on the rating and review length. Here's some meta-code to get you thinking about this problem:

Sentiment data generation `sentimentr( review_text )`

Loop a linear model of sentiment `lm( sentiment ~ rating + length )`

Abstract over the results `colMeans(bootstrapped_results)`

Part 6: Final thoughts

I don't often write loops, but when I do, they run in parallel.

To get the most out of parallel loops, you will need to have access to as many cores (processors) as possible. Consider using remote computing resources, such as Quest (Northwestern). For the best performance, use vectorized code whenever possible, such as the functions provided in `tidyverse` and `data.table`, rather than loops.

That said, many processes are easy to write as loops because the code runs sequentially, which is easy to conceptualize. Rewriting loops using vectorized functions essentially gives you more power as a researcher, because it unlocks the computing power you already have. More importantly, *it saves your time!*

Happy looping looping looping!

One last step

Best practice would be to close the script with stopping the cluster…

```
# This severs the connection between R and the 'extra' cores we were leveraging.
stopCluster( cl )
```