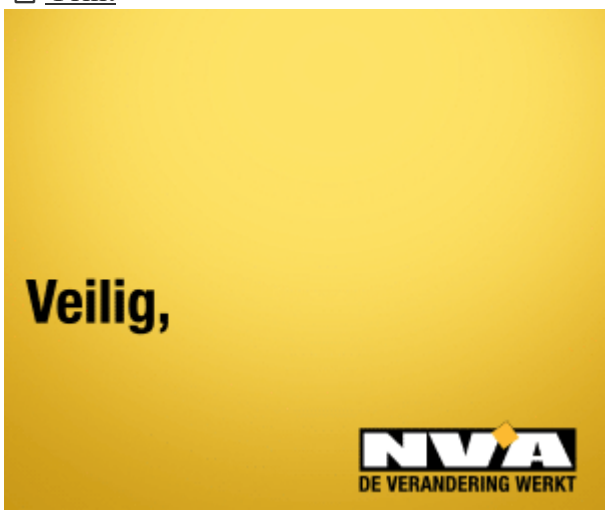
Programming

Strategies to Speedup R Code

- Published on January 30, 2016 at 1:56 pm
- Updated on May 30, 2018 at 7:32 pm



The for-loop in R, can be very slow in its raw un-optimised form, especially when dealing with larger data sets. There are a number of ways you can make your logics run fast, but you will be really surprised how fast you can actually go.
This posts shows a number of approaches including simple tweaks to logic design, parallel processing and `Rcpp`, increasing the speed by orders of several magnitudes, so you can comfortably process data as large as 100 Million rows and more.

I am going to show you the various approaches using an example logic that involves a for-loop and a condition checking statement (if-else) to create a column that gets appended to a sufficiently large data frame (df). Lets begin by creating that initial dataframe.

```
# Create the data frame
col1 <- runif (12^5, 0, 2)
col2 <- rnorm (12^5, 0, 2)
col3 <- rpois (12^5, 3)
col4 <- rchisq (12^5, 2)
df <- data.frame (col1, col2, col3, col4)Copy
```

The logic we are about to optimise:
For every row on this data frame (df), check if the sum of all values is greater than 4. If it is, a new 5th variable gets the value "greater_than_4", else, it gets "lesser_than_4".

```
# Original R code: Before vectorization and pre-allocation
system.time({
  for (i in 1:nrow(df)) { # for every row
    if ((df[i, "col1"] + df[i, "col2"] + df[i, "col3"] + df[i, "col4"]) > 4) { # check if > 4
      df[i, 5] <- "greater_than_4" # assign 5th column
    } else {
      df[i, 5] <- "lesser_than_4" # assign 5th column
    }
  }
})Copy
```

All the computations below, for processing times, were done on a MAC OS X with 2.6 Ghz processor and 8GB RAM.

Vectorise and pre-allocate data structures

Always initialise your data structures and output variable to required length and data type before taking it to loop for computations. Try not to incrementally increase the size of your data inside the loop. Lets compare how vectorisation improves speed on a range of data sizes from 1000 to 100,000 rows.

```
# after vectorization and pre-allocation
output <- character (nrow(df)) # initialize output vector
system.time({
  for (i in 1:nrow(df)) {
    if ((df[i, "col1"] + df[i, "col2"] + df[i, "col3"] + df[i, "col4"]) > 4) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- "lesser_than_4"
    }
  }
df$output})Copy
```

Raw Code Vs With vectorisation:

Take statements that check for conditions (if statements) outside the loop

Taking the condition checking outside the loop the speed is compared against the previous version that had vectorisation alone. The tests were done on dataset size range from 100,000 to 1,000,000 rows. The gain in speed is again dramatic.

```
# after vectorization and pre-allocation, taking the condition checking outside the loop.
output <- character (nrow(df))
condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4  # condition check outside the loop
system.time({
  for (i in 1:nrow(df)) {
    if (condition[i]) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- "lesser_than_4"
    }
  }
  df$output <- output
})Copy
```

Condition Checking outside loops:

Run the loop only for True conditions

Another optimisation we can do here is to run the loop only for condition cases that are ‘True’, by initialising (pre-allocating) the default value of output vector to that of ‘False’ state. The speed improvement here largely depends on the proportion of ‘True’ cases in your data.

The tests compared the performance of this against the previous case (2) on data size ranging from 1,000,000 to 10,000,000 rows. Note that we have increase a ‘0’ here. As expected there is a consistent and considerable improvement.

```
output <- character(nrow(df))
condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4
system.time({
  for (i in (1:nrow(df))[condition]) {  # run loop only for true conditions
    if (condition[i]) {
      output[i] <- "greater_than_4"
    } else {
      output[i] <- "lesser_than_4"
    }
```

```
  }
  df$output })Copy
```

Running Loop Only On True Conditions:

Use ifelse() whenever possible

You can make this logic much simpler and faster by using the `ifelse()` statement. The syntax is similar to the `if` function in MS Excel, but the speed increase is phenomenal, especially considering that there is no vector pre-allocation here and the condition is checked in every case. Looks like this is going to be a highly preferred option to speed up simple loops.

```
system.time({
  output <- ifelse ((df$col1 + df$col2 + df$col3 + df$col4) > 4, "greater_than_4", "lesser_than_4")
  df$output <- output
})Copy
```

True conditions only vs ifelse:

Using which()

By using `which()` command to select the rows, we are able to achieve one-third the speed of `Rcpp`.

```
# Thanks to Gabe Becker
system.time({
  want = which(rowSums(df) > 4)
  output = rep("less than 4", times = nrow(df))
  output[want] = "greater than 4"
})
# nrow = 3 Million rows (approx)
   user  system elapsed
  0.396   0.074   0.481
Copy
```

Use apply family of functions instead of for-loops

Using apply() function to compute the same logic and comparing it against the vectorised for-loop. The results again is faster in order of magnitudes but slower than `ifelse()` and the version where condition checking was done outside the loop. This can be very useful, but you will need to be a bit crafty when handling complex logic.

```
# apply family
system.time({
  myfunc <- function(x) {
    if ((x['col1'] + x['col2'] + x['col3'] + x['col4']) > 4) {
      "greater_than_4"
    } else {
      "lesser_than_4"
    }
  }
  output <- apply(df[, c(1:4)], 1, FUN=myfunc)  # apply 'myfunc' on every row
  df$output <- output
})Copy
```

Use byte code compilation for functions cmpfun() from compiler package, rather than the actual function itself

This may not be the best example to illustrate the effectiveness of byte code compilation, as the time taken is marginally higher than the regular form. However, for more complex functions, byte-code compilation is known to perform faster. So you should definitely give it a shot.

```
# byte code compilation
library(compiler)
myFuncCmp <- cmpfun(myfunc)
system.time({
  output <- apply(df[, c (1:4)], 1, FUN=myFuncCmp)
})Copy
```

apply vs for-loop vs byte code compiled functions:

Use Rcpp

Lets turn this up a notch. So far we have gained speed and capacity by various strategies and found the most optimal one using the `ifelse()` statement. What if we add one more zero? Below we execute the same logic but with `Rcpp`, and with a data size is increased to 100 Million rows. We will compare the speed of `Rcpp` to the `ifelse()` method.

```
library(Rcpp)
sourceCpp("MyFunc.cpp")
system.time (output <- myFunc(df)) # see Rcpp function belowCopy
```

Below is the same logic executed in C++ code using Rcpp package. Save the code below as “MyFunc.cpp” in your R session’s working directory (else you just have to sourceCpp from the full filepath). Note: the `// [[Rcpp::export]]` comment is mandatory and has to be placed just before the function that you want to execute from R.

```
// Source for MyFunc.cpp
#include
using namespace Rcpp;
// [[Rcpp::export]]
CharacterVector myFunc(DataFrame x) {
  NumericVector col1 = as(x["col1"]);
  NumericVector col2 = as(x["col2"]);
  NumericVector col3 = as(x["col3"]);
  NumericVector col4 = as(x["col4"]);
  int n = col1.size();
  CharacterVector out(n);
  for (int i=0; i 4){
      out[i] = "greater_than_4";
    } else {
      out[i] = "lesser_than_4";
    }
  }
  return out;
}Copy
```

`Rcpp` speed performance against `ifelse`:

Use parallel processing if you have a multicore machine

Parallel processing:

```
# parallel processing
library(foreach)
library(doSNOW)
cl <- makeCluster(4, type="SOCK") # for 4 cores machine
registerDoSNOW (cl)
condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4
# parallelization with vectorization
system.time({
  output <- foreach(i = 1:nrow(df), .combine=c) %dopar% {
    if (condition[i]) {
      return("greater_than_4")
    } else {
      return("lesser_than_4")
    }
  }
})
df$output <- outputCopy
```

Remove variables and flush memory as early as possible

Remove objects `rm()` that are no longer needed, as early as possible in code, especially before going in to lengthy loop operations. Sometimes, flushing `gc()` at the end of each iteration with in the loops can help.

Use data structures that consume lesser memory

`Data.table()` is an excellent example, as it reduces the memory overload which helps to speed up operations like merging data.

```
dt <- data.table(df)  # create the data.table
system.time({
  for (i in 1:nrow (dt)) {
    if ((dt[i, col1] + dt[i, col2] + dt[i, col3] + dt[i, col4]) > 4) {
      dt[i, col5:="greater_than_4"]  # assign the output as 5th column
    } else {
      dt[i, col5:="lesser_than_4"]  # assign the output as 5th column
    }
  }
})Copy
```

Dataframe Vs Data.Table:

Speed Summary

Method: Speed, nrow(df)/time_taken = n rows per second
Raw: 1X, 120000/140.15 = 856.2255 rows per second (normalised to 1)
Vectorised: 738X, 120000/0.19 = 631578.9 rows per second
True Conditions only: 1002X, 120000/0.14 = 857142.9 rows per second
ifelse: 1752X, 1200000/0.78 = 1500000 rows per second
which: 8806X, 2985984/0.396 = 7540364 rows per second
Rcpp: 13476X, 1200000/0.09 = 11538462 rows per second

The numbers above are approximate and are based in arbitrary runs. The results are not calculated for `data.table()`, byte code compilation and parallelisation methods as they will vary on a case to case basis, depending upon how you apply it.

Author

Selva Prabhakaran
Data Scientist

More from Author

- Outlier detection and treatment with R
- Chi-Squared Test – The Purpose, The Math, When and How to Implement?
- Missing Value Treatment

Disclosure

- Selva Prabhakaran does not work or receive funding from any company or organization that would benefit from this article. Views expressed here are personal and not supported by university or company.

Trending Now on DataScience+

Clean Your Data in Seconds with This R Function

How to write the first for loop in R

Aggregate – A Powerful Tool for Data Frame in R

How to Perform a Logistic Regression in R

Building A Logistic Regression in Python, Step by Step

Tags AlgorithmData FramesData ManipulationTips & Tricks
Discussion

22 Comments **DataScience+ Hub** **Login**

Recommend 2 **Share** **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS**

Name

João Neto • 3 years ago

The apply solution can be simplified:

system.time({

df$output <- apply(df, 1, myfunc) # apply 'myfunc' on every row

})

ps: the browser seems to be confusing some code chars as HTML, that's why some code is not appearing. There's always CTRL+U's view source :-)

23 • Reply • Share ›

Klodian Mod → João Neto • 3 years ago

I think, all code is correctly showing now?

1 • Reply • Share ›

Brian Stamper • 3 years ago

Either some of the code is not rendering right in this post or I'm just not understanding the shorthand. For example,

after vectorization and pre-allocation

output 4) [etc..]

What is that "output 4)"? Or in later sections we see "condition 4". Something isn't right here..

1 • Reply • Share ›

Selva Prabhakaran → Brian Stamper • 3 years ago

Yes, you are right. Something is not right with the code rendering. This will be rectified asap. Thanks for pointing out.

• Reply • Share ›

Michail Unknown • a year ago

For those interested try the R package Rfast and also have a look at this document
http://rfast.eu/site/templa...

Reply • Share ›

Jesús Pacheco • 2 years ago

Great article! I haven't seen anyone mention logical indexing; it takes only 0.06secs on my 6GB computer to process 2M rows. I do believe there's an effect of diminishing returns (so to speak) as the number of rows increase, I guess because we need to store a vector of the same size with the corresponding logical values. This is just a guess, but my PC did crash when I tried to process 100M rows!

I couldn't get the Rcpp example to run: I do notice that the #include <rcpp.h> and the 'if' statements are missing, so I tried something like this:

```
// Source for MyFunc.cpp
#include <rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
CharacterVector myFunc(DataFrame x) {
NumericVector col1 = as(x["col1"]);
NumericVector col2 = as(x["col2"]);
NumericVector col3 = as(x["col3"]);
NumericVector col4 = as(x["col4"]);
```

see more

Reply • Share ›

rcodeprogramming ➚ Jesús Pacheco • 2 years ago

The code rendering broke again. I will notify the admin!

Reply • Share ›

Jesús Pacheco ➚ rcodeprogramming • 2 years ago

awesome, thanks

Reply • Share ›

ItsMe • 2 years ago

#for checking conditions out of the loop, what if i am referencing two different rows? In the loop i currently use below code in the nested loop.
if (dat[i,3]=!dat[j,3])

#i do not think that i can replace it with below
condition <- dat[,3]!=dat[,17]

#because i cannot specify 'j' in the if condition

if condition(1) {
#do task
}

Do you have a way out for this?

Reply • Share ›

John Butters • 3 years ago

As a finance practitioner I found this incredibly interesting and useful. Thank you!

Reply • Share ›

Selva Prabhakaran ➚ John Butters • 3 years ago

Welcome!

Reply • Share ›

Chris Kypridemos • 3 years ago

Overall nice examples
However, your example using data.table is inefficient.
A most appropriate data.table way would be

```
require(data.table)
setDT(df)
df[, output := "greater_than_4"]
df[(col1 + col2 + col3 + col4) <= 4, output := "less_than_4"]
```

Which I bet will match or even outperform the RCPP example you have provided

Reply • Share ›

Selva Prabhakaran ➚ Chris Kypridemos • 3 years ago

Thanks, I will review this.

Reply • Share ›

Taylor White • 3 years ago

These are extremely contrived examples. Under no circumstances should one use a for loop to define individual elements of a vector or data.frame as a constant using a loop.

Just use ifelse! Where output is a data.frame:
output$col_name = ifelse(something > 4, 'greater_than_4', 'le_than_4')

The examples on this page exhibit bad coding practice by using unnecessary loops. Instead, show examples of things that HAVE to be done in a loop and demonstrate how different looping strategies can have a big effect.

A beginner might find this page and think "oh, I should use Rcpp to really speed up defining elements of a vector!"

Reply • Share ›

rcodeprogramming ➚ Taylor White • 2 years ago

It shows how the same task can be done with a number of approaches, improving the performance step after step. The case of ifelse is also covered.

1 • Reply • Share ›

Mark Adamson • 3 years ago

When comparing apply and for loops, the text suggests that apply is better, but the graph shows apply as much slower. Is the labelling wrong?

• Reply • Share ›

Selva Prabhakaran ➚ Mark Adamson • 3 years ago

Yes, my mistake, Thanks for pointing this out :)

• Reply • Share ›

Bert Bertrand • 3 years ago

Very useful information! You may enjoy this article, your post reminded me of it --> http://journals.plos.org/pl...

• Reply • Share ›

Sound Advice • 3 years ago

Great article! Thank you!

• Reply • Share ›

Selva Prabhakaran ➚ Sound Advice • 3 years ago

Welcome :)

• Reply • Share ›

flodel • 3 years ago

Nice overview. I think you can improve the which() approach by doing c("greater_than_4", "less_than_4")[1L + ((df[[1]] + df[[2]] + df[[3]] + df[[4]]) <= 4)]. In particular, doing the sum this way is faster than using rowSums which has to convert your data to a matrix first. At this point, I think this is all using C under the hood so it should be comparable to Rcpp.

• Reply • Share ›

Selva Prabhakaran ➚ flodel • 3 years ago

Very nice :) Shall add this method as well.

• Reply • Share ›

Understanding Titanic Dataset with H2O's AutoML, DALEX, …

16 comments • a month ago

BarcaDad — try dfm = train instead. there is no df data frame in his code

Accessing Web Data (JSON) in R using httr

1 comment • 2 months ago

vaibhav kulkarni — Awesome article written by akshay 👍👍 definitely it's helpful for those …

DataScience+Bridging the gap between talent and opportunity

›_ Site Links

- About Us
- Archives
- Contribute
- R Markdown

›_ Legal

- Privacy Policy
- Terms of Use
- Account Terms
- Contact Us

›_ Articles

- Introduction
- Getting Data
- Data Management
- Visualizing Data
- Basic Statistics
- Regression Models
- Advanced Modeling
- Programming
- All Articles →

Connect with Us

datascience+ jobs