FasteR! HigheR! StrongeR! - A Guide to Speeding Up R Code for Busy People

(From: http://noamross.net/blog/2013/4/25/faster-talk.html)

by *Noam Ross*, 25 April 2013

This is an overview of tools for speeding up your R code that I wrote for the Davis R Users' Group[1].

First, Ask "Why?"

It's customary to quote Donald Knuth[2] at this point, but instead I'll quote my twitter buddy Ted Hart to illustrate a point:

> I'm just going to say it.I like for loops in #Rstats[3], makes my code readable.All you [a-z]*ply snobs can shove it!
>
> — Ted Hart (@DistribEcology) March 12, 2013[4]

Code optimization is a matter is a matter of personal taste and *priorities*. There may be some ways of writing code that are better or worse, and there are definitely ways that will make it run faster, but before you dive into optimization, you should ask yourself these questions:

(1) Do you want your code to be *readable*?

If you need to explain your code to yourself or others, or you will need to return to it in a few months time and understand what you wrote, it's important that you write it in a way that is easy to understand. Comments, indenting, and formatting go a long way, but your choices make a difference, as Ted notes above.

Some optimal code can be hard to read. For instance, and curly braces can be faster than parentheses. But which is more intuitive to read?[1] [5]

(2) Do you want your code to be *sharable*?[2] [6]

Most of the considerations of (1) apply here, but they have to be balanced with the fact that, if your code is painfully slow, others are not going to want or have time to use it.

Also, some optimization strategies don't transfer well. If you use parallel processing, it won't be usable to others who don't have multicore computers. Some tricks for speeding up code will work well as a quick-and-dirty fix, but are likely to break in new versions of R or are disallowed from CRAN packages altogether.

(3) Do you have other shit to do?

No? Please contact me and I'll help you with that. Yes? You are among the 90% or R users who's first priority is not computer programming. The time spent optimizing code is often longer than the computing time actually saved. Use the simple solutions and you can get on with your research/life.

With these in mind, I'll go through some strategies for speeding up your R computations. Because I'm especially mindful of consideration #3, this guide is somewhat in the reverse of traditional tutorials on this topic. I'm going to talk about the blunt instruments first, then get into the idiosyncrasies of code.

Get a Bigger Computer

If you have a task that only needs to be accomplished a few times, you might be best off finding more resources rather than re-writing your code. You have an Amazon.com account, right? Amazon rents virtual machines on the web. It's so easy to set one up that I accidentally started one today without meaning to while searching for the link. The cost of computing power roughly equivalent to my 2011 MacBook Pro is *13¢ per hour*. You can have 100 MacBooks of computing power for less than minimum wage[7].[3][8] These virtual machines can also have much more memory than your laptop or desktop.

Sometimes you don't even need more computing power, you just need to run R on some other computer without tying up your machine for hours. In any case, consider this option. If you just need to run a task once, and you think you are going to have to spend more than an hour rewriting your code, you might as well use that hour getting a virtual machine set up and never worry about computing power again.

Cloud resources like Amazon aren't the only way to get more computer power. If you're at a university, there are probably high-power computing clusters available to you. Depending on your particular set-up, you may need to to run your program in batch mode[9] to use such clusters. Personally, I think that a cloud-based service like Amazon avoids administrative hassle if you are on a short deadline or have a one-off task, but if you need lots of power regularly, getting in on a local cluster is worthwhile.

The most user-friendly way to use cloud computing is to set up a server to run RStudio, which can then be accessed by a web browser and works just like it would on your desktop. Bioconductor has a step by step guide[10] to starting an Amazon Web Instance that they've pre-configured this way. There are a couple of other useful guides to doing the same thing by Louis Aslett (including a video)[11] and Karthik Ram[12]. Read all of these and you should be able to set yourself up in an hour or two.

A fancier alternative is the experimental Segue package[13], which provides an `apply()` function variant, `emrapply()`, that sends commands from your local R session to Amazon servers and returns the results. An introductory guide is here[14].

Of course, if you are writing software that you expect others to run on normal laptop and desktop computers, you'll want to make sure it does so at a reasonable speed. In this case, running your software in the cloud may not make sense.

Parallel Processing

Parallel processing used to be a very fancy thing, but with recent R packages, it can be implemented with very little programming time. Parallel processing breaks up your task, splits it among multiple processors, and puts the components back together. This is very useful and easy if you have a task that *can* be split up, especially without the different parts needing to "talk to" each other.

On your typical laptop or desktop, implementing parallel processing in your R code might speed up your program by a factor of 2 to 4. However, if you have a big machine, or you followed my advice in the last section, you can take advantage of large computers with many CPUs or "cores".

I'm not going to get into the nitty gritty of setting up parallel processes and the various packages that do so e.g. `multicore`, `snow`,`snowfall`, `doParallel`,`foreach`, `plyr`,...) because Jacob Teter[15] will give a presentation on that at D-RUG[1] next week. Instead I'll discuss a bit about *when* this approach might be appropriate.

Essentially, parallel processing works best when you have a task that has to be completed many times, but each repeat is independent. For instance, if you are repreating a simulation, and in each case are drawing new parameters from a distribution, that is an easily parallelizable task. However, if you are simulating population dynamics, in which each time step you need the outputs of the previous time steps, than you can't split up each time step into parallel components.

A good rule of thumb is that if you can wrap your task in an `apply` function or one of its variants, it's a good option for parallelization. In fact most implementations of parallel processing in R are versions of `apply`.

Not every task runs better in parallel. When you run a task in parallel, your computer "dispatches" each task to a CPU core. This dispatching adds computational overhead. So, it's usually best to try to minimize the number of dispatches. In most cases you are going to have a small number of computing cores relative to your tasks. A powerful laptop or desktop will have 2, 4, or 8 cores, and even the most powerful Amazon virtual machine has 88.

The following would be a poor use of parallel computing:

```
aaply(1:10000, 1, function(x) rnorm(1, mean=x), .parallel=TRUE)
```

This would dispatch the very small task of generating a random number 10,000 times. The dispatching would probably take more time than the actual calculations. Instead, organize the tasks into blocks, and dispatch each block of tasks, for instance:

```
aaply(seq(1,10000,100), function(x) rnorm(1, mean=x+(1:100), .parallel=TRUE)
```

This sends the task of generating 100 numbers to each core, reducing the overhead.

[Update 4/26/2013: As Vince Buffalo points out[16], random number generation requires that R keep track of state, so may not be a good example of a parallel-izable task]

In practice, this means that you want to "parallelize" your code at the highest, or chunkiest level. If you have a series of nested loops, for instance, you will want to run the highest-level loop in parallel.

Note that more cores is not always better[17]. For reasons I don't completely understand, the overhead of using more cores is not always worth it, so it may be better to use, say 16, rather than 64 codes. Some experimentation will help here.

One other thing: if you can limit the parallel processes to computational tasks, and avoid things like reading/writing to disk or downloading files in the parallel code, you'll be less likely to run into problems.

A disadvantage of parallel computing is that it is not available to everyone. People run code on different machines. If you are writing software that you expect to share with others (and as a scientist, you should ALWAYS be prepared to share your work on request, and post your code as a supplement to your paper),

you will want to provide an option to run it not in parallel. I like to do something like this:

```
#This script uses parallel processing if p.flag=TRUE.  Set up a parallel
#cluster as appropriate for your machine as appropriate. (the commented code
#below will use 2 cores on a multicore computer)
#library(doParallel)
#cl <- makeCluster(2)  # Use 2 cores
#registerDoParallel(cl) # register these 2 cores with the "foreach" package
library(plyr)
p.flag=FALSE  # Change to TRUE if using parallel processing
.
.
.
 aaply(seq(1,10000,100), function(x) rnorm(1, mean=x+(1:100),
      .parallel=p.flag)
```

Compiling Functions

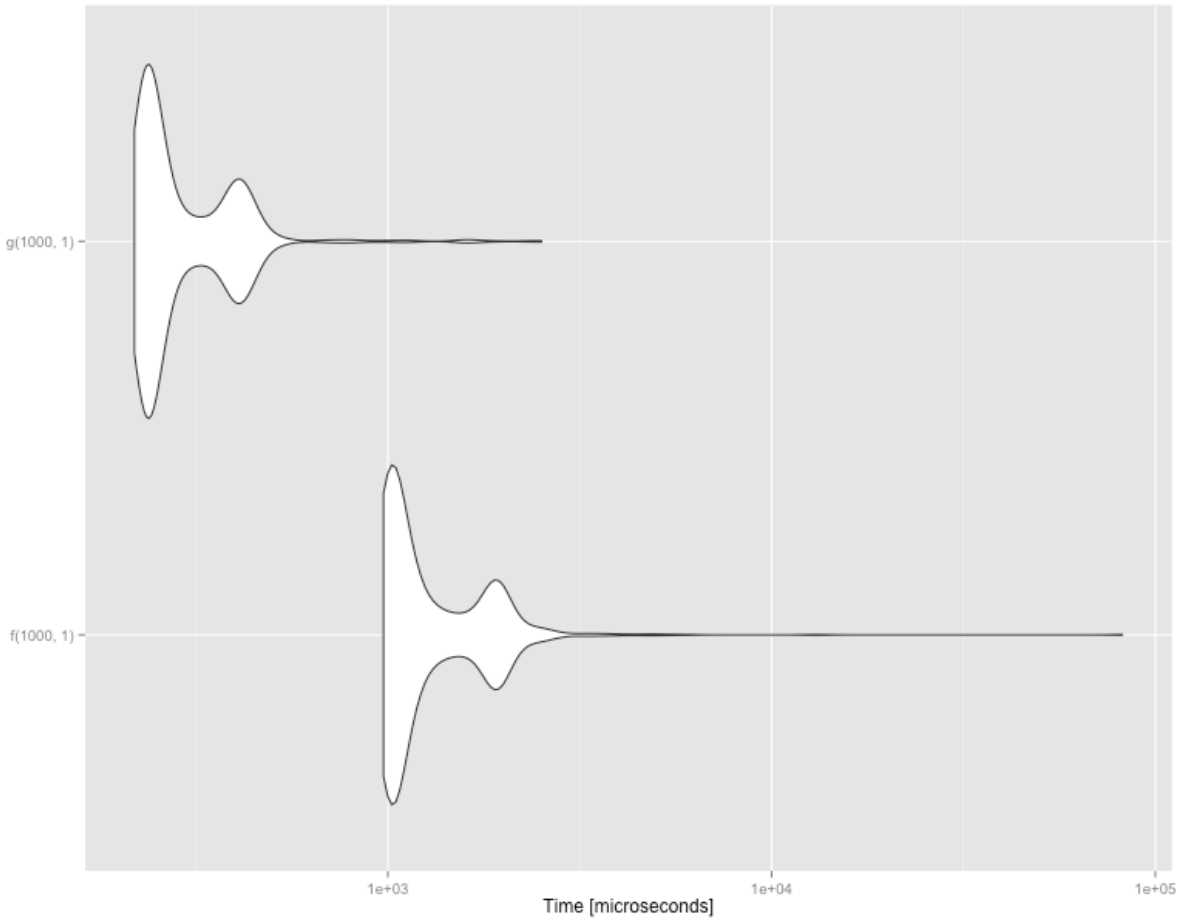
R code is interpreted when it is run, unlike some other languages, which are first compiled. This is one reason why functions written in C are often faster than functions written in R.

However, R has compiling ability, which can speed up functions by a factor or 3 or 4 *in some cases*[18]. Compiling functions just requires calling `cmpfun()` in the base `compiler` package. Here's an example:

```
library(compiler)
f <- function(n, x) for (i in 1:n) x = (1 + x)^(-1)
g <- cmpfun(f)

library(microbenchmark)
compare <- microbenchmark(f(1000, 1), g(1000, 1), times = 1000)

library(ggplot2)
autoplot(compare)
```



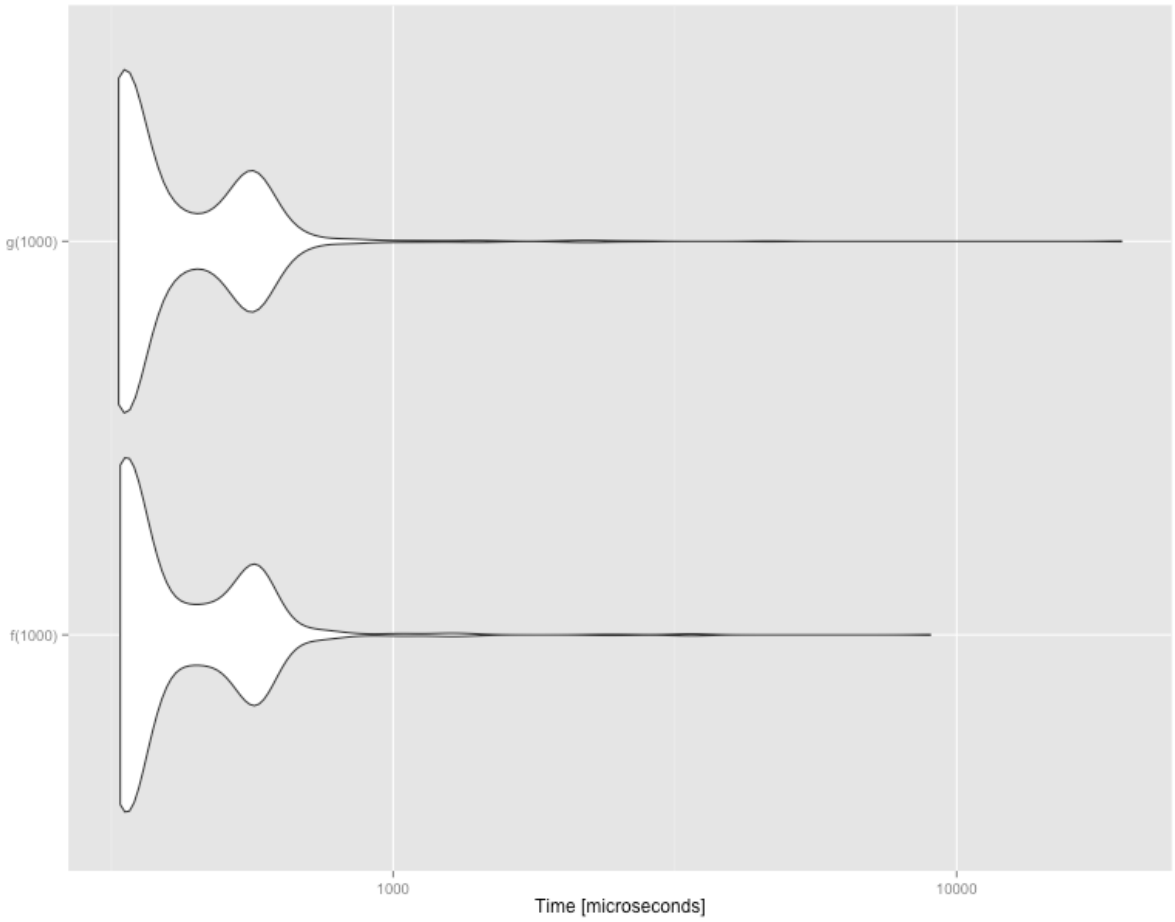

```
compare

## Unit: microseconds
##        expr   min    lq median     uq  max neval
##  f(1000, 1) 950.0 991.3 1024.7 1087.2 2799  1000
##  g(1000, 1) 218.2 228.0  238.2  257.3 1743  1000
```

(See the benchmarking section below for more on comparing functions with `microbenchmark()`)

`cmpfun()` works well in cases like this - when you are defining a new function that is mostly numerical manipulation. It doesn't help with functions that call a lot of other R functions, or involve manipulating and translating between data types. It also doesn't work well on already-defined R functions, which have been pre-compiled since R 2.14 and are sometimes written in compiled languages like C. For instance, if I try to compile this `paste()`-based function, I get no improvement:

```
f <- function(n) paste(1:n, collapse = " ")
g <- cmpfun(f)
compare <- microbenchmark(f(1000), g(1000), times = 1000)
autoplot(compare)
```




```
compare

## Unit: microseconds
##     expr   min    lq median    uq  max neval
##  f(1000) 320.2 331.8  353.9 429.2 1889  1000
##  g(1000) 320.7 330.6  352.8 412.5 2530  1000
```

You can also enable just-in-time compiling in R[19], which will automatically compile EVERY function the first time that it is run. This will slow down R quite a bit at first, as each function must be compiled before it is run the first time, but but then speed it up later. Just add this to the start of your script

```
library(compiler)
enableJIT(1)
```

The numeric argument in `enableJIT` specifies the "level" of compilation. At level 2 and 3 more functions are compiled. In a few rare cases enableJIT will *slow down* code, particularly, if most of it is already pre-compiled or written in C, and/or if the code *creates* functions repeatedly, which then need to be compiled every time. This is more likely to happen with `enableJIT(2)` or `enableJIT(3)`, though these have the potential to speed up code more, as well.

Diagnosis: Profiling and Benchmarking

OK, the tools I described above are all blunt instruments that could speed up many types of code. However, if they fail you or aren't feasible, you're going to need to know something about what is slowing your code down.

Benchmarking

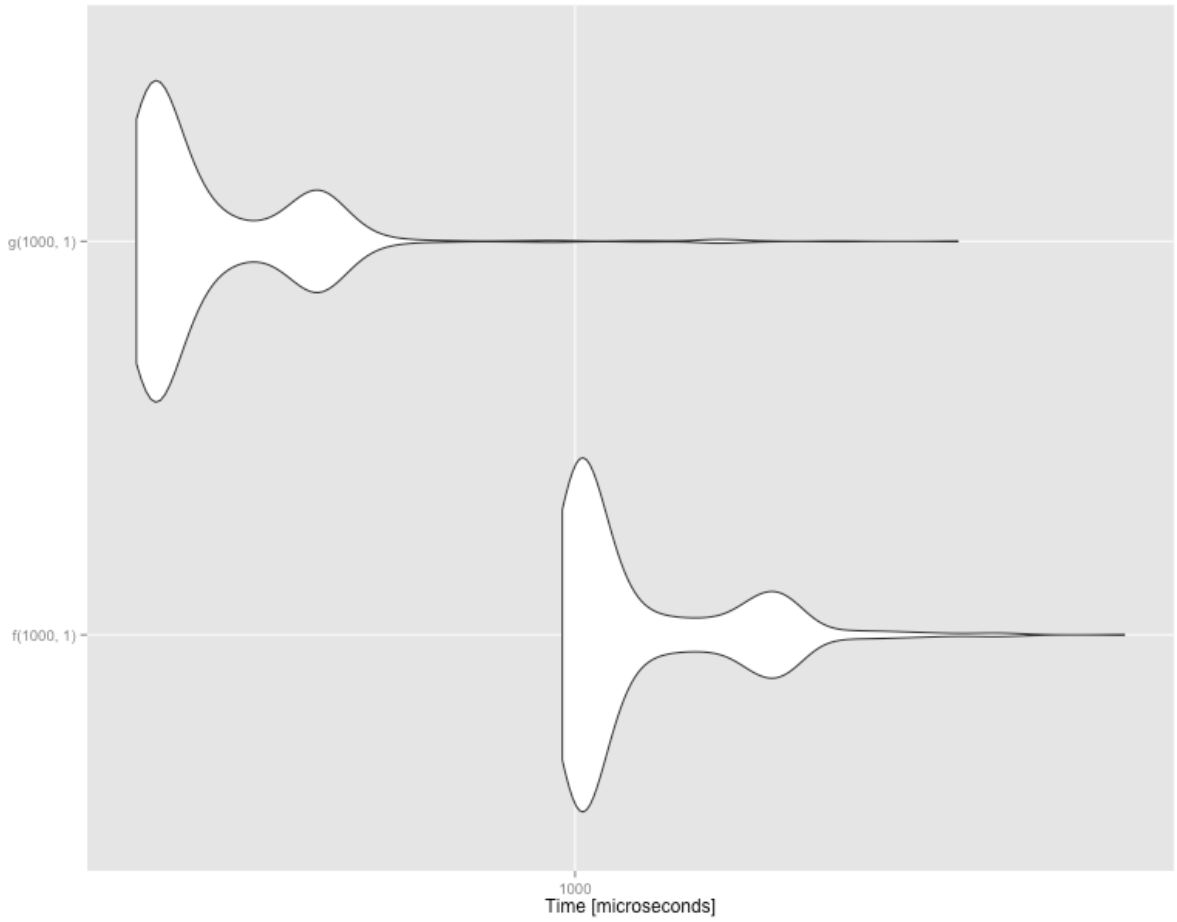
Benchmarking is a way to test and compare the speed of functions.

Traditionally, `system.time(replicate())` could be used to time functions, but the `microbenchmark` package is a better option for benchmarking individuals calls and functions. If you have identified a function that is called many times in your code and needs to be speeded up, you can write or try out several different versions, using `microbenchmark` to compare them. Here's the example from above:

```
f <- function(n, x) for (i in 1:n) x = (1 + x)^(-1)
g <- cmpfun(f)

library(microbenchmark)
compare <- microbenchmark(f(1000, 1), g(1000, 1), times = 1000)

library(ggplot2)
autoplot(compare)
```




```
compare

## Unit: microseconds
##        expr   min     lq median     uq   max neval
##  f(1000, 1) 958.2 1064.2 1220.5 1795.7 11595  1000
##  g(1000, 1) 219.8  243.7  271.4  379.4  1370  1000
```

All `microbenchmark` needs as inputs are a series of expressions to compare. `times` is an optional parameter of the number of times to test each function (default 100). Microbenchmark randomizes the test order and discards some "warmup" tests. The full distribution is returned as a data frame, but only the summaries are printed.

The `autoplot()` command draws a comparison of distribution times using `ggplot()`

Profiling

Benchmarking is good for comparing small functions. *Profiling* examines your code to determine what parts of it are running slow. This is the essential tool for getting the best bang for your buck in speeding up code.

R's code profiler samples the "call stack" at regular intervals. The call stack is the list of the current function running, the function that called it, and all the functions that *those*. By examining these samples, you can find

which functions are taking up the largest amount of time, either because they are slow or are called many times.

You activate the profiler with `Rprof("filename")` which writes each sample of the call stack to "filename". So do this:

```
Rprof("file.out")
Some_slow_code()
Rprof(NULL)
summaryRprof("file.out")
```

`Rprof(NULL)` stops sampling and `summaryRprof("file.out")` gives you a summary of the results. Here's an example of profiling a random walk function (from a talk by Duncan Temple Lang[20]):

```
rw2s1 <- function(n) {
    xpos = ypos = numeric(n)
    xdir = c(TRUE, FALSE)
    pm1 = c(1, -1)
    for (i in 2:n) if (sample(xdir, 1)) {
        xpos[i] = xpos[i - 1] + sample(pm1, 1)
        ypos[i] = ypos[i - 1]
    } else {
        xpos[i] = xpos[i - 1]
        ypos[i] = ypos[i - 1] + sample(pm1, 1)
    }
    list(x = xpos, y = ypos)
}

Rprof("out.out")
for (i in 1:1000) pos = rw2s1(1000)
Rprof(NULL)
summaryRprof("out.out")

## $by.self
##              self.time self.pct total.time total.pct
## "sample.int"      7.62    47.04       7.62     47.04
## "rw2s1"           5.28    32.59      16.20    100.00
## "sample"          2.90    17.90      10.52     64.94
## "-"               0.30     1.85       0.30      1.85
## "+"               0.04     0.25       0.04      0.25
## "list"            0.04     0.25       0.04      0.25
## "numeric"         0.02     0.12       0.02      0.12
##
## $by.total
##              total.time total.pct self.time self.pct
## "rw2s1"           16.20    100.00      5.28    32.59
## "sample"          10.52     64.94      2.90    17.90
## "sample.int"       7.62     47.04      7.62    47.04
## "-"                0.30      1.85      0.30     1.85
```

```
## "+"                0.04      0.25      0.04      0.25
## "list"             0.04      0.25      0.04      0.25
## "numeric"          0.02      0.12      0.02      0.12
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 16.2
```

The two tables list the functions called. The `$by.self` component lists the time spent by *functions alone*, while the `$by.total` table lists the time spent by *functions and all the functions they call*. So you see that `rw2s1`, being the *parent function*, takes the most time in `$by.total`, but `sample.int` is the bottom-level function that is taking the most time.

I don't find this `summaryRprof()` output very intuitive. There are some moderately better tools out there in the packages `profr` and `proftool`, but I don't like them either. Instead I've written a custom function that you can get here[21] called `proftable`. Here it parses the same output as above:

```
proftable("out.out")
```

```
##  PctTime Stack
##  47.037  rw2s1 > sample > sample.int
##  32.593  rw2s1
##  17.901  rw2s1 > sample
##   1.852  rw2s1 > -
##   0.247  rw2s1 > +
##   0.247  rw2s1 > list
##   0.123  rw2s1 > numeric
##
## Total Time: 16.2 seconds
```

Here you can see that >50% of time is being taken up by `sample.int`, which is itself usually called by `sample`.

Misc profiling things

- The latest version of R (3.0.0) allows code profiling by *line number* using `Rprof(file, line.profiling=TRUE)` and `summaryRprof(file, lines="show")`. I don't yet have a good example of it and haven't incorporated this function into `proftable()` yet. There's an example of use at Stack Overflow[22]
- Note that, depending on the R interface you are using, it may wrap your code in some other functions, which will appear as the topmost functions in the stack. For instance, if use the "Source" button in Rtudio, all your stacks will start with something like:

```
source > withVisible > eval > eval >
```

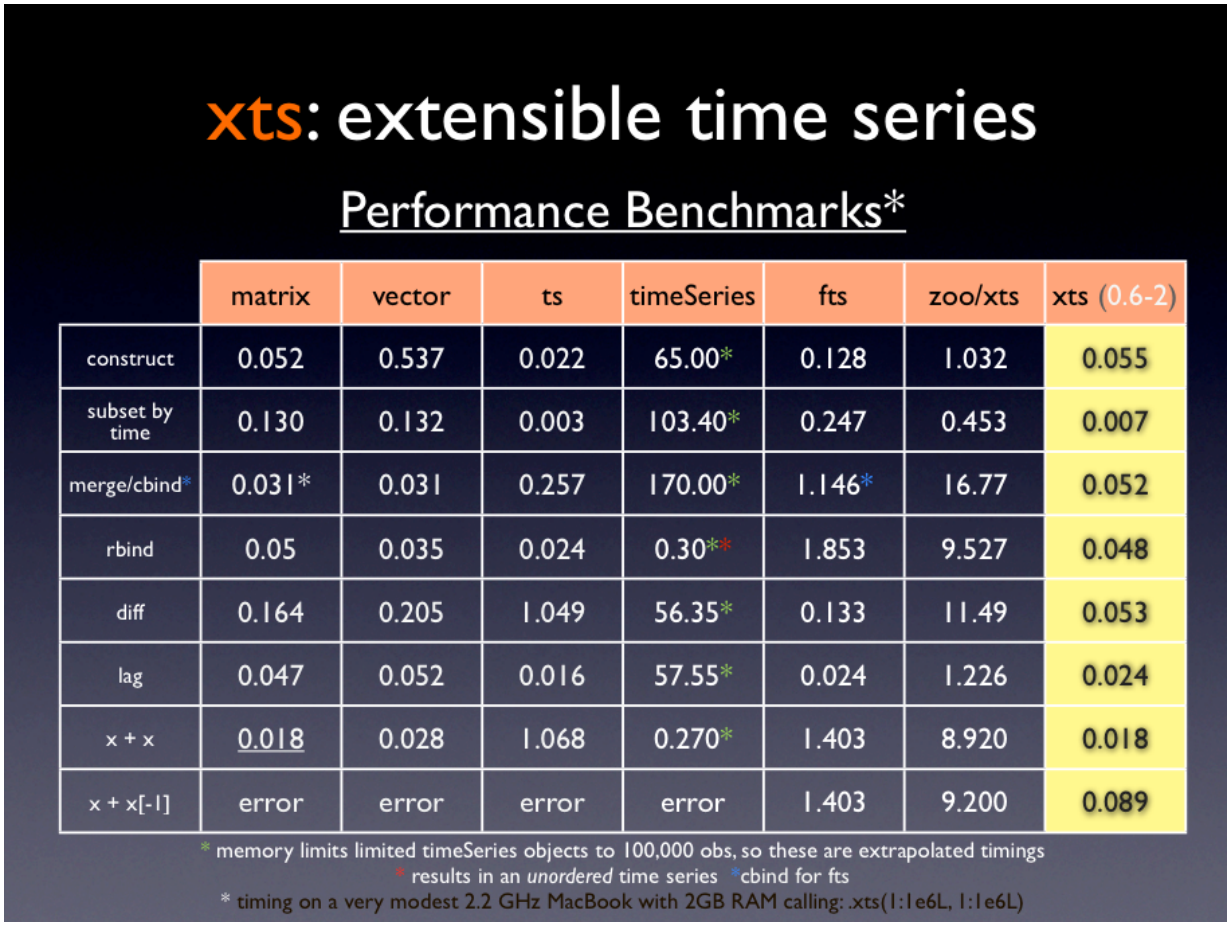
[Update 5/8/2003: The latest verion[21] of `proftable` can deal with line numbers. It also deals with wrapper stacks like above.]

- `trace()` can also be useful to track down where a particular slow function is being called. Try using `trace(function, tracer=cat(as.character(sys.calls()))`, for instance, to print the stack whenever `function` is called.
- R also has the ability to do *memory profiling* - tracking how your code uses memory. These include `Rprofmem()`, `tracemem()`, and `Rprof(memory.profiling=TRUE)`, but the documentation for these isn't great and I haven't found good resources on this topic. Let me know if you can point me to some resources.
- Jeromy Anglim has a blog post of an `Rprof()` case study[22][3] that is worth reading

Find better packages

Before you go about changing your own code, take a look and see if you can use better code someone else has written. There are thousands of packages on CRAN, but many of them are redundant and implement the same thing. There's tremendous variation in package quality and speed, so you can often speed things up by finding a faster package.

For instance, the `xts` and `zoo` packages both handle time series data, but much of `xts` is written in C and Fortran, and it is considerably faster. Here are some performance benchmarks (though they are a bit out of date):



Comparison of functions on xts time series objects against time series from other packages. From this presentation[24]

Other packages are specifically designed to speed up certain operations. Some creative googling (e.g., *r package CRAN fast "something"*) will help you find them.

For instance. Using base functions like `colMeans()` is faster than using `apply` functions to apply `mean`. The `matrixStats` package extends these functions to include others like `rowMaxs()`, `colRanks()`, etc.

`data.table` is a package designed for speeding up data frame operations when working very large datasets. `data.table`'s biggest advantage is in tasks like sub-setting and indexing. Here I compare the two in looking up the value of a row:

```
library(data.table)
myDF <- as.data.frame(matrix(rnorm(1e+05), 10000, 10))
myDF$key <- as.character(1:nrow(myDF))
myDT <- data.table(myDF)
setkey(myDT, key)
microbenchmark(myDF[myDF$key == 2, ], myDT["2", ])

## Unit: microseconds
##                   expr     min      lq median    uq   max neval
```

```
##  myDF[myDF$key == 2, ] 2658.8 2885.3   3187 3755 6434   100
##            myDT["2", ]  724.3  899.7   1112 1393 4287   100
```

Here `data.table` is 3X times faster. It gets faster with larger data sets.

Improving your code

Once you know what parts of your code are slowing you down, and you have a method to test speed, you need tools and options for writing faster functions. Here are a few.

Vectorization

This is the first thing most guides to coding discuss, so I won't go into in much. Basically, R is good at matrix/vector algebra, so if you can express something as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

It will go faster than

$$\begin{aligned} x + a \\ y + b \\ z + c \end{aligned}$$

Or, in R

```
x <- c(1,2,3) + c(1, 2, 3)
```

Instead of

```
for(i in 1:3) x[i] <- i+i
```

- Most base R functions are designed to handle vectors as well as scalars.
- However, using an `apply` function is not really vectorization. `apply` functions are actually wrappers for `for` loops
- For more on vectorization, see Chapters 3 and 4 of The R Inferno[25]

Pre-allocating memory

Again, a common topic I won't dwell on too much. Essentially, R is bad at continually re-sizing objects, because it makes an extra copy of these objects each time. So if you have a loop that creates a vector or list, don't append to the vector or list with each pass of the loop. Instead, make an *empty* object of the correct size first, then fill in its elements.

More here[26]

The `apply` family of functions helps you avoid these problems by doing pre-allocation automatically. Plus,

as I wrote above, if you use `apply` variants, it's easy to parallelize your code.

Using simpler data structures

R has some very handy objects, lists and dataframes, that are flexible and can store multiple types of data at once. However, this flexibility comes at a cost. Simpler data structures that only store *one* type of data can be manipulated much faster. If you can represent data in a matrix instead of a data frame (e.g., if you can reduce it to all numbers or characters), you can speed things up considerably.

For instance, compare these operations for getting the standard deviation of 100,000 rows of 10 variables in both a matrix and a data frame:

```
myMatrix <- matrix(rnorm(1e+05), 10000, 100)
myDF <- as.data.frame(myMatrix)
microbenchmark(apply(myMatrix, 2, sd), apply(myDF, 2, sd))

## Unit: milliseconds
##                   expr   min    lq median    uq    max neval
##  apply(myMatrix, 2, sd) 29.70 31.60  33.04 34.65  94.21   100
##      apply(myDF, 2, sd) 39.69 70.14  96.61 98.81 250.12   100
```

The small stuff

Little things can add up, but the little things are most likely not your main problem. Here are a few odds and ends I've discovered. They'll shave a few percent off your run time, but thinking about them too much will add hours to your coding time.

Extra parentheses

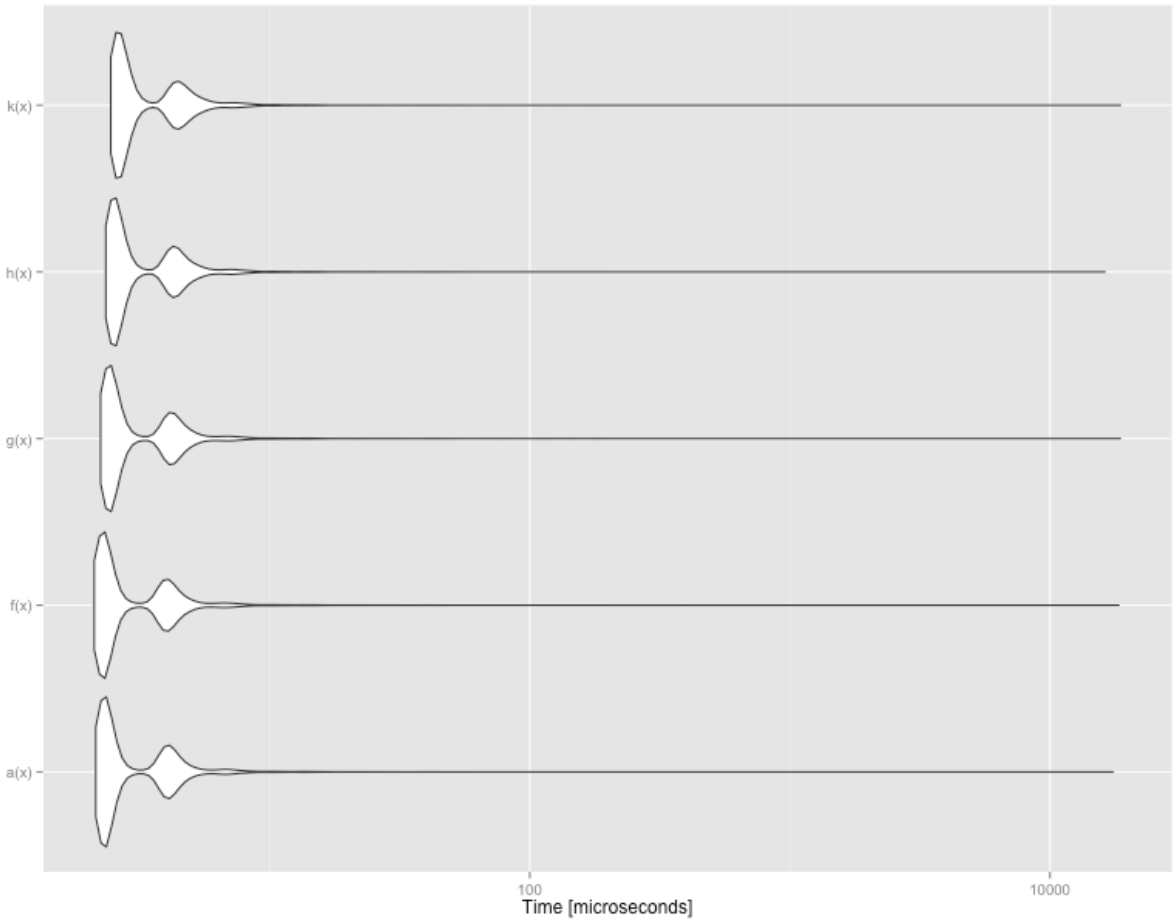
Sometimes when I write complex mathematical expressions I include extra parentheses to help make the code more readable. This has a small performance cost, as you can see here:

```
a <- function(x) x = 1/{
    1 + x
}
f <- function(x) x = 1/(1 + x)
g <- function(x) x = (1/(1 + x))
h <- function(x) x = ((1/(1 + x)))
k <- function(x) x = (((1/(1 + x))))
x <- sample(1:100, 100, replace = TRUE)
comp <- microbenchmark(a(x), f(x), g(x), h(x), k(x), times = 1e+05)
comp

## Unit: microseconds
##  expr   min    lq median    uq   max neval
##  a(x) 1.991 2.197  2.313 2.971 11068 1e+05
##  f(x) 1.960 2.174  2.292 2.990 10435 1e+05
##  g(x) 2.090 2.302  2.427 3.112 10755 1e+05
```

```
##  h(x) 2.188 2.406  2.536 3.178  9660 1e+05
##  k(x) 2.259 2.504  2.644 3.367 11228 1e+05

autoplot(comp)
```



The difference is modest, but each layer of parentheses adds a bit to performance time. Note that, amazingly, r speeds up when you use curly braces (`{` and `}`) instead of parentheses! Why? Radford Neal explains[27]:

> The difference between parentheses and curly brackets comes about because R treats curly brackets as a "special" operator, whose arguments are not automatically evaluated, but it treats parentheses as a "built in" operator, whose arguments (just one for parentheses) are evaluated automatically, with the results of this evaluation stored in a LISP-style list. Creating this list requires allocation of memory and other operations which seem to be slow enough to cause the difference, since the curly bracket operator just evaluates the expressions inside and returns the last of them, without creating such a list.

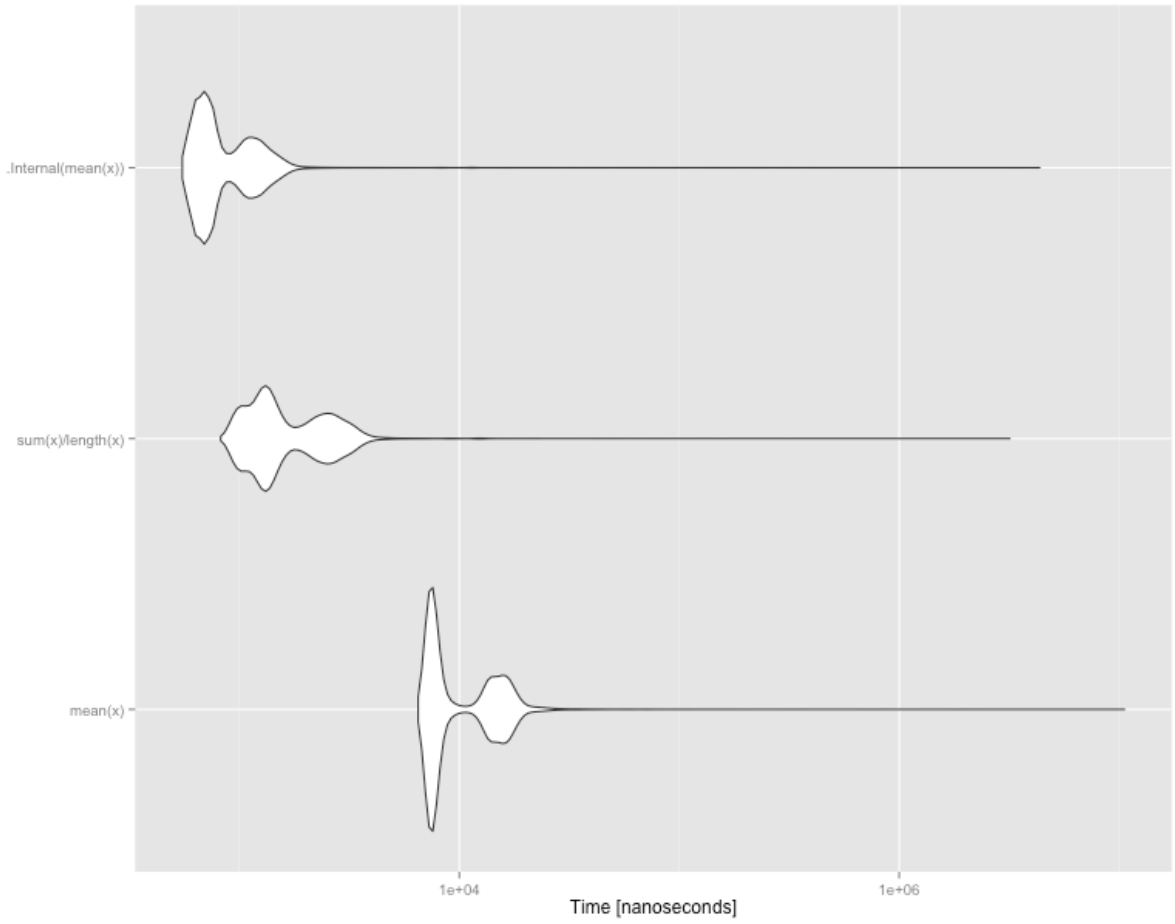
Standard and internal functions

A few standard functions in R are pretty slow, often because they perform error checking and methods dispatch before actually doing arithmetic. `mean` is among the most notorious of these. `mean(x)` does a lot more than just `sum(x)/length(x)`. It checks the form of the inputs, assigns the appropriate method, then dispatches `.Internal(mean(x))`. The latter is much faster, as it is written in C:

```
x <- sample(1:100, 100, replace = TRUE)
comp <- microbenchmark(mean(x), sum(x)/length(x), .Internal(mean(x)), times = 1e+05)
comp

## Unit: nanoseconds
##                 expr  min   lq median   uq      max neval
##              mean(x) 5671 6136   6386 6792 61466725 1e+05
##    sum(x)/length(x)  720  878   1064 1211  1511792 1e+05
##  .Internal(mean(x))  513  556    614  677  1082112 1e+05

autoplot(comp)
```



You can sometimes get away with using the internal functions like `.Internal(mean(x))`. However, this is considered poor practice because (a) internals are subject to change and break your code with updates to R, and (b) this is disallowed in packages submitted to CRAN, so it makes code more difficult to share. That said,

the internal version can be 9X faster. Your call.

[Update 4/26/2013: On the suggestion of Kevin Ushey[28], I'm adding the following section]

Other Languages: More reward for more effort

I've mentioned a few times that functions written in other languages, like C or C++, are much faster than other R functions. Obviously, learning another language is an undertaking, but thanks to some recent developments, writing R functions in C has become much easier. The `Rcpp` package lets you write such C++ functions without learning much about compiling or other peripheral issues, and Hadley Wyckham has written an excellent beginners's guide to `Rcpp`[29] that teaches you just enough C++ to get started. From the guide:

> Typical bottlenecks that C++ can help with are:
>
> - Loops that can't easily be vectorised because each iteration depends on the previous. C++ modifies objects in place, so there is little overhead when modifying a data structure many times.
> - Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than the overhead of calling a function in R. To give you some idea of the magnitude, on my computer when writing this book the overhead in C++ was ~5ns compared to ~200ns for R.
> - Problems that require advanced data structures and algorithms that R doesn't provide. Through the standard >template library (STL), C++ has efficient implementations of many important data structures, from ordered maps >to double ended queues.

---

1. I used to also have a reference here that `x*x` is faster than `x^2` in R, but as Thomas Lumley points out[30], this hasn't been true for a while, and some quick empirical testing confirmed that the two are pretty close.↩[31]
2. If you are a scientist, you should ALWAYS be prepared to share your code, and should probably publish it as a supplement to your papers.↩[32]
3. There are cheaper services than Amazon, but there aren't as many resources on how to use them. If you really need to reduce costs, you can use *spot instances*[33], which allow you to set an allowable price, and run your program when something that cheap is available. Also, you could run many small Amazon virtual machines simultaneously on a free trial account, but if your goal is to do things *quickly*, it might be worth paying for a more powerful machine.↩[34]

---

1. http://www.noamross.net/davis-r-users-group.html
2. http://en.wikiquote.org/wiki/Donald_Knuth
3. https://twitter.com/search/%23Rstats

4. https://twitter.com/DistribEcology/status/311581025870426113
5. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fn1
6. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fn2
7. http://greentheo.scroggles.com/2009/11/13/amazon-ec2-providing-100-macbooks-of-power-for-minimum-wage/
8. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fn3
9. http://stat.ethz.ch/R-manual/R-patched/library/utils/html/BATCH.html
10. http://bioconductor.org/help/bioconductor-cloud-ami/
11. http://www.louisaslett.com/RStudio_AMI/
12. http://inundata.org/2011/03/30/r-ec2-rstudio-server/
13. https://code.google.com/p/segue/
14. http://jeffreybreen.wordpress.com/2011/01/10/segue-r-to-amazon-elastic-mapreduce-hadoop/
15. http://www.linkedin.com/pub/jacob-teter/9/7b2/684
16. https://twitter.com/vsbuffalo/status/327564597282369536
17. http://lookingatdata.blogspot.se/2013/01/speeding-up-r-computations-pt-iii.html
18. http://dirk.eddelbuettel.com/blog/2011/04/12/#the_new_r_compiler_package
19. http://www.r-statistics.com/2012/04/speed-up-your-r-code-using-a-just-in-time-jit-compiler/
20. http://www.stat.auckland.ac.nz//~ihaka/downloads/Taupo-handouts.pdf
21. https://github.com/noamross/noamtools/blob/master/R/proftable.R
22. http://stackoverflow.com/a/15821706/1757441
23. http://jeromyanglim.blogspot.com/2010/02/case-study-in-optimising-code-in-r.html
24. http://www.quantmod.com/Columbia2008/ColumbiaDec4.pdf
25. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
26. http://musicallyut.blogspot.com/2012/07/pre-allocate-your-vectors.html
27. http://radfordneal.wordpress.com/2010/08/19/speeding-up-parentheses-and-lots-more-in-r/
28. https://twitter.com/kevin_ushey/status/327205714328158208
29. https://github.com/hadley/devtools/wiki/Rcpp
30. http://simplystatistics.org/2013/04/28/sunday-datastatistics-link-roundup-4282013/#comment-879142735
31. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fnref1
32. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fnref2
33. http://aws.amazon.com/ec2/spot-instances/
34. http://www.noamross.net/blog/2013/4/25/faster-talk.html#fnref3

Tags: R D-RUG

digdeep • 8 months ago

Hi Noam, great post. I was wondering how you go about actually getting performance gains with R for jamming a large file in memory and are just wanting to crunch a giant table faster, not necessarily looking at repeated iterations and going down the parallel path. As I understand it, R only uses part of your memory but I am unsure how you can override this (if at all possible) to allocate more memory easily for certain tasks then drop it back to normal. Is this something you can do easily in R? or better, RStudio?

Noam Ross Mod ↗digdeep • 8 months ago

Check out the `data.table` package. Data.table provides an alternative structure for data frames that is backed by a binary database, and is MUCH faster for operations on big tables. The beta package `dplyr` (https://github.com/hadley/dply... implements functions from plyr on data.tables.

The package `bigmemory` and its complements such as `bigmatrix` create matrix-type objects that are stored as files on the disk or in shared RAM space in a more efficient way. I haven't used it myself yet.

digdeep ↗Noam Ross • 8 months ago

Yep, an avid user of data.table. Wouldn't be able to function without this package. Trying to test out packages like btyd on a million or so customer files, things tend to take a long time. With these scripts, however long it takes will probably be quicker than recoding it all. So looking for a sort of brute force option at first, if not, then a more long term approach is necessary I guess. Will check out bigmemory and bigmatrix.

Rinat Menyashev • 11 months ago

Thanks! Very insightful!! After switching to R from Stata, I was really surprised how things slowed down

ronert_obst • a year ago

Very informative post!

Jesse Fagan • a year ago

Ooh, ooh, do one for memory management!

Noam Ross Mod ↗Jesse Fagan • a year ago

Man, I'd love to if I knew anything about it. The documentation for memory profiling is *terrible* and I can't find any good resources on it. Let me know if you

discover anything.

Reply • Share ›

Ken • a year ago

In many ways the use of C or C++ is the most important. What it means is that I can write my R code in a way that is readable without trying tricks involving vectorization that are difficult to read. Then look at the important loops and recode in C. You wont have to do much and leaves you with readable code.

Reply • Share ›

buggyfunbunny • a year ago

-- I like for loops in #Rstats, makes my code readable. All you [a-z]*ply snobs can shove it!

Well, sounds like a 20-something who never groked: prolog, set theory, relational model, and relational databases.

The point of the apply verbs is superior semantics. Sometimes, one gets efficiency. One learns early on that iteration belongs on the metal, not in user code. That coders of limited ability gravitate to iteration as panacea says more about them than the alternatives. Iteration is a crutch, not track shoes.

Reply • Share ›

BarryR • a year ago

You've totally forgotten something vital. Write Tests. Otherwise you don't know if your speedups haven't introduced new bugs. This step is NOT optional imho.

Reply • Share ›

Richard Kwo • a year ago

Wow, nice post. I am also a grad student interested in stat stuff. I just wonder, what tool do you use to make your neatly-typesetted blog?

Reply • Share ›

Noam Ross Mod ↗Richard Kwo • a year ago

I use jekyll to generate my blog. The styling is mostly custom CSS. More here: https://github.com/noamross/je...

Reply • Share ›

someone • a year ago

Shouldn't that be StrongeR? Beautiful post BTW.

Reply • Share ›

Noam Ross Mod ↗someone • a year ago

D'oh! Thanks.

Reply • Share ›

Ted Underwood • a year ago

I hope you'll also post, or tweet, a link to Jacob Teter's presentation on parallelization in R.

• Reply • Share ›

Noam Ross Mod ↗Ted Underwood • a year ago

Will do. I post all the Davis R Users' Group presentations: http://www.noamross.net/tags.h...

• Reply • Share ›

Ted Underwood • a year ago

Bookmarked. I really appreciate the way you organized this in probable order of importance.

• Reply • Share ›

Robert Norberg • a year ago

Awesome post! This is going in my bookmarks for future reference.

• Reply • Share ›