# High-Performance Computing in R

## Introduction to Computational BioStatistics with R

Alexey Fedoseev

November 27, 2025

## High-performance R

Just a reminder:

- R is an interpreted language. As such, there is an extra layer of infrastructure (the interpreter) needed to make R run
- As a general rule, because of the extra layer of infrastructure, interpreted languages (R, Python, Bash, Perl, . . . ) are not high-performance languages
- True high-performance languages are compiled, and thus they lack the extra layer of infrastructure: C, C++, Fortran
- That being said, there are ways of making R better. That is the goal of this class

## R and memory

One must be cognisant of how R manages memory:

- R is “pass-by-value” if the variables being passed are being modified within the function. As such, R frequently needs to make temporary copies of variables, and hitting the memory limit of your machine can be a frequent problem
- Like many dynamic languages, R relies on “garbage collection” to limit its memory usage
- In a running program, “every so often” a garbage collection task runs and deletes variables that won’t be used any more
- You can force the garbage collector to run at any given time by calling `gc()`, but this almost never fixes anything significant
- How can GC know that you’re not going to use that big variable in the next line? The garbage collector needs your help to be effective

## Useful memory-management commands

- `gc(verbose = TRUE)`, or just `gc(TRUE)`
  - Calling `gc(TRUE)` alone probably won't help anything, but it does give verbose output, returning memory usage as a matrix
- `ls()`
  - Lists all existing variables, as strings
- `object.size(variablename)`
  - Pass it a variable, and it prints out its size
  - Pass it get("variablename") and it will also print its size
- `rm(variablename)`
  - Deletes a variable you no longer need. Lets gc go to work
- Fun little one-liner which prints out all variables by size in bytes

```
> sort(sapply(ls(), function(x) {object.size(get(x))}), decreasing = TRUE)
```

## object.size and gc

```
> gc()
          used  (Mb) gc trigger  (Mb)  max used   (Mb)
Ncells  183250   9.8     407500  21.8     350000  18.7
Vcells  377223   2.9     905753   7.0     864975   6.6
> get.mem <- function() return(gc()[, 1:2])
> old.mem <- get.mem()
> x <- rep(0., (16 * 1024)**2)
> xsize <- object.size(x)
> xsize
2147483688 bytes
> print(xsize, units = "MB")
2048 Mb
> get.mem() - old.mem
             used   (Mb)
Ncells        445      0
Vcells  268436139   2048
```

## object.size and gc, some more

Now let's delete the object and see how the system memory behaves

```
> rm(x)
> final.mem <- get.mem()
> final.mem - old.mem
         used  (Mb)
Ncells    451   0.1
Vcells   1781   0.0
```

Be sure to delete temporary variables in your scripts, especially large ones!

## Profiling

To push your code to new heights of awesome, or to make it useful at all (depending on your situation), you will need to profile your code. What is profiling?

- Profiling is analyzing where the code is spending its time. Which parts of the code are slowest?
- Testing how long individual functions take can be performed with the ‘microbenchmark’ package, or more crudely, `system.time`
- To test the whole program we use the `Rprof` function

We’ll do some examples of each.

## Profiling individual functions

- The `system.time` command uses the OS's `time` command to determine how long the code takes to run.

```
> f <- function() {
+   a <- 1
+   for (i in 1:1e8) {
+     a <- a + i
+   }
+ }
> system.time(f())
   user  system elapsed
  0.964   0.002   0.967
```

## Profiling individual functions - `microbenchmark`

- The `microbenchmark` function is more systematic. It takes an average over 100 calls of the function. Consequently, it can take a while to run
- Note that the microbenchmark package will need to be downloaded

```
> microbenchmark(f(), times=10)
Unit: milliseconds
 expr     min      lq     mean   median       uq      max neval
  f() 927.771 929.782 939.4129 940.1643 943.3133 952.1787    10
```

## Profiling individual functions - `microbenchmark`

- You can use `microbenchmark` to compare performance of multiple functions:

```
x = runif(100)
microbenchmark(
  sqrt(x),
  x ^ 0.5
)
```

```
Unit: nanoseconds
    expr  min   lq    mean median     uq   max neval
 sqrt(x)  287  328  604.34    369  430.5 10742   100
   x^0.5 1968 2009 2106.17   2050 2091.0  5248   100
```

## Profiling whole programs

Use `Rprof` to analyse where the code is spending its time.

```
> addme <- function(a, b) { Sys.sleep(0.001); return(a + b) }
> test <- function() {
+   a <- 1
+   for (i in 1:1e5)
+     a <- addme(a, i)
+ }
> Rprof("Rprof.data")
> test()
> Rprof(NULL)
> s <- summaryRprof("Rprof.data")
> s$by.total
                           total.time total.pct self.time self.pct
"test"                           1.82     100.0      0.00      0.0
"Sys.sleep"                      1.80      98.9      1.80     98.9
"addme"                          1.80      98.9      0.00      0.0
```

## Rprof

Some notes about the last slide:

- `Rprof` samples the program every 20ms, by default, to see where the program is spending its time
- Use `Rprof("filename")` to store the `Rprof` results in a particular file
- Use `Rprof(NULL)` to turn off profiling
- You can read “filename” if you want. It’s easier to just use `summaryRprof("filename")` to analyse the results
- Results are given in data frames
- Columns `total.time` and `total.pct` (total percent) include all time spent within a function, including calls to other functions
- `self.time` and `self.pct` indicate actual time spent in each function (`self.pct` should add up to 100%, give or take rounding).

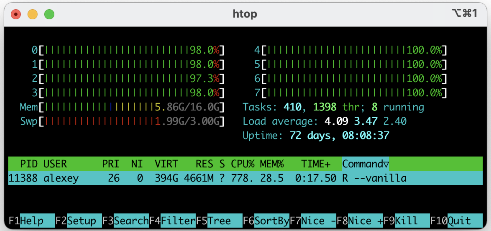
## Existing parallelism

It’s important to realize that many fundamental routines as well as higher-level packages come with some degree of scalability and parallelism “baked in”.

Open another terminal to your node, and run “top” while executing the following in R:

```
>
> n <- 4 * 1024
>
> A <- matrix( rnorm(n * n), ncol = n, nrow = n )
> B <- matrix( rnorm(n * n), ncol = n, nrow = n )
>
> C <- A %*% B
>
```

## Existing parallelism



```
    0[|||||||||||||||||||||||||98.0%]     4[||||||||||||||||||||||||100.0%]
    1[|||||||||||||||||||||||||98.0%]     5[||||||||||||||||||||||||100.0%]
    2[|||||||||||||||||||||||||97.3%]     6[||||||||||||||||||||||||100.0%]
    3[|||||||||||||||||||||||||98.0%]     7[||||||||||||||||||||||||100.0%]
  Mem[|||||||||||||||||||5.86G/16.0G]   Tasks: 410, 1398 thr; 8 running
  Swp[|||||||||||||||||||1.99G/3.00G]   Load average: 4.09 3.47 2.40
                                        Uptime: 72 days, 08:08:37

  PID USER      PRI  NI  VIRT   RES S CPU% MEM%   TIME+  Command▽
11388 alexey     26   0  394G 4661M ? 778. 28.5  0:17.50 R --vanilla


F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

One R process using 778% of a processor.

R can (and should) be built using high-performance threaded libraries for math in general, and linear algebra in particular.

Here the single R process has launched several threads of execution – all of which are part of the same process, and so can see the same memory.

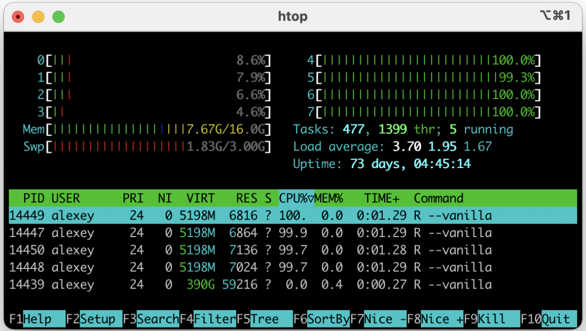
## `mclapply`

One simple way to parallelize your code is to use `mclapply`, which works the same way as `lapply`, but forking off the processes (forking does not work on Windows):

```
> library(parallel)
> add.me <- function(n) {
+   a <- 1
+   for (i in 1:n) a <- 1 / (1 + a)
+ }
> system.time(list.res <- lapply(rep(1e8,4), add.me))
   user  system elapsed
  5.532   0.007   5.539
> system.time(list.par.res <- mclapply(rep(1e8,4), add.me, mc.cores = 4))
   user  system elapsed
  4.495   0.013   1.516
```

## mclapply

Note what the output of top looks like when this is running:



```
   0[|||                     8.6%]   4[||||||||||||||||||||||||100.0%]
   1[|||                     7.9%]   5[|||||||||||||||||||||||||99.3%]
   2[|||                     6.6%]   6[||||||||||||||||||||||||100.0%]
   3[||                      4.6%]   7[||||||||||||||||||||||||100.0%]
 Mem[||||||||||||||||||||7.67G/16.0G]  Tasks: 477, 1399 thr; 5 running
 Swp[|||||||||||||||||||1.83G/3.00G]  Load average: 3.70 1.95 1.67
                                      Uptime: 73 days, 04:45:14

  PID USER       PRI  NI  VIRT   RES S CPU%▽MEM%   TIME+  Command
14449 alexey      24   0 5198M  6816 ? 100.  0.0  0:01.29 R --vanilla
14447 alexey      24   0 5198M  6864 ? 99.9  0.0  0:01.29 R --vanilla
14450 alexey      24   0 5198M  7136 ? 99.7  0.0  0:01.28 R --vanilla
14448 alexey      24   0 5198M  7024 ? 99.7  0.0  0:01.29 R --vanilla
14439 alexey      24   0  390G 59216 ?  0.0  0.4  0:00.27 R --vanilla

F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

There are multiple processes running - not one process using multiple CPUs via threads.

## foreach - serial

The standard R `for` loop looks like this:

```
> for (i in 1:2) print(sqrt(i))
[1] 1
[1] 1.414214
```

The `foreach` operator looks similar (you may need to install the `foreach` package), but returns a list of the iterations:

```
> library(foreach)
> foreach (i=1:2) %do% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214
```

## foreach + doParallel

Foreach works with a variety of backends to distribute computation.

Switching the above loop to parallel just requires registering a backend and using %dopar% rather than %do%:

```
> library(doParallel) # make sure doParallel package is installed
> registerDoParallel(3) # use forking, does not work on Windows
> foreach (i=1:2) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

> stopImplicitCluster()
```

## foreach + doParallel

One can also use a PSOCK cluster:

```
> cl <- makePSOCKcluster(3) # works on Windows
> registerDoParallel(cl) # use the just-made PSOCK cluster
> foreach (i=1:3) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

> stopCluster(cl)
```

## Combining results

While returning a list is the default, `foreach` has a number of ways to combine the individual results:

```
> foreach (i=1:3, .combine=c) %do% sqrt(i)
[1] 1.000000 1.414214 1.732051
> foreach (i=1:3, .combine=cbind) %do% sqrt(i)
     result.1 result.2 result.3
[1,]        1 1.414214 1.732051
> foreach (i=1:3, .combine="+") %do% sqrt(i)
[1] 4.146264
> foreach (i=1:3, .multicombine=TRUE, .combine="sum") %do% sqrt(i)
[1] 4.146264
```

Most of these are self-explanatory. `multicombine` is worth mentioning: by default, `foreach` will combine each new item individually. If `.multicombine=TRUE`, then you are saying that you're passing a function which will do the right thing even if foreach gives it a whole chunk of new results as a list or vector - *e.g.*, a whole chunk at a time.

## Running across multiple computers

You can run your code on multiple computers by specifying them to the function `makePSOCKcluster`:

```
> library(foreach)
> library(doParallel)
> hosts <- rep(c("tri-login01", "tri-login02"), 2)
> cl <- makePSOCKcluster(names=hosts, outfile="")
starting worker pid=4163978 on tri-login01:11494 at 19:24:39.138
...
> registerDoParallel(cl)
> foreach (i=seq_along(hosts), .combine=data.frame) %dopar%
+    c(Sys.info()["nodename"], "Process ID"=Sys.getpid(), calc=i**2)
              result.1    result.2    result.3    result.4
nodename   tri-login01 tri-login02 tri-login01 tri-login02
Process ID     4165542     3136113     4165641     3136230
calc                 1           4           9          16
> stopCluster(cl)
```

## Parallel RNG

Depending on what you are doing, it may be very important to have different (or the same!) random numbers generated in each process.

`parallel` has a good RNG suitable for parallel work based on the work of Pierre L'Ecuyer in Montréal:

```
> RNGkind("L'Ecuyer-CMRG")
> mclapply(rep(1,2), rnorm, mc.cores=2, mc.set.seed=TRUE)
[[1]]
[1] -0.4982475

[[2]]
[1] 0.9267458
```

## Compiled code

It is possible to interface your R code with compiled code. Why would you want to do that?

- It's fast! Compiled code is always faster than interpreted code
- If you can get the slowest parts of your code into a compiled language, you can leave the rest in R
- R comes with the ability to byte-compile specific functions
- It's also possible to write your own pure C++ or Fortran code to interface with R, but it's a pain
- It's easier to use the Rcpp package, written by Dirk Eddelbuettel, Romain Francois, and others
- This package allows you to easily interface with C++ code

## Byte-compiled R code

We can byte-compile specific R functions using the `compiler` package. Since R 3.4.0, loops are automatically byte-compiled before they are run, and all functions are compiled on their first or second use.

Here we're using the `enableJIT` (Just In Time compiler) function to turn off automatic byte compiling. In general, you should NOT do this. We're only doing this for the purposes of comparing speeds.

```
> library(compiler); library(microbenchmark)
> oldJIT <- enableJIT(0); n <- 1e5
> f <- function(n) { x <- 1; for (i in 1:n) x <- 1 / (1 + x) }
> lf <- cmpfun(f)
> microbenchmark(f(n), lf(n))
Unit: milliseconds
  expr       min        lq      mean    median       uq       max neval
  f(n) 13.439841 13.778255 14.309054 13.860767 14.38052 27.337693   100
 lf(n)  1.359232  1.362942  1.372936  1.364152  1.37596  1.526307   100
```

## Byte-compiled R code

Some notes about the last slide:

- Byte compiling is not the same as actually compiling code, as is done with compiled languages:
    - Byte compiling creates a byte object, which is executed by a virtual machine
    - Compiled languages are compiled into machine code, which is directly used by the hardware
- Nonetheless, byte compiling can be significantly faster than running the code through the R interpreter
- If you run a function multiple times, R will automatically byte-compile it for you. Better to just byte-compile it in your utilities file.
- Automatic byte compiling can be turned off using the `enableJIT` function, though this is not recommended

## Installing `Rcpp`

We're going to be doing examples with `Rcpp`. But, if you're using Windows...

- `Rcpp` is not a default R package; you will need to download and install it
- Because `Rcpp` compiles code (that's the point), you will need a compiler on your computer
- If you're using Linux or a Mac, you're probably OK
- On Windows, you need to go here, and download "Rtools":

https://cran.r-project.org/bin/windows/Rtools

Note that Rtools is quite large, and will require some time to download. It's probably best not to do this during class.

## Using Rcpp

Once the function is defined, it will automatically be compiled, this is why it takes a moment for the cppFunction command to finish.

Once compiled, Rcpp creates an R function which links to the compiled C++ code.

```
> library(Rcpp)
> cppFunction("int times(int x, int y) {
+   int product = x * y;
+   return product;
+ }")
> times(34, 4)
[1] 136
> 34 * 4
[1] 136
```

## Using `Rcpp`

Some notes about this example:

- `Rcpp` defines special C++ data types which are compatible with R data types:
  - IntegerVector, NumericVector, LogicalVector, CharacterVector
  - IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix
  - Lists, DataFrames
- These data types allow the ability to deal with missing values, using the `is_na()` function

Note that you should always test your code carefully when using multiple languages. Sometimes surprises can creep in.

## Making your code awesome

Some tips:

- Save your function profiling until you know that the function works correctly. Don't succumb to "premature optimization"
- Do byte compiling first. It's easy and may be good enough
- Put your byte-compiled functions in your utilities files
- Don't be afraid of `Rcpp`. Once you know how to program in one language, you're at least 80% of the way to programming in all languages
- Ask us for help, if speed becomes an issue for your productivity