High Performance Computing with R

Eric Feigelson (Penn State) edf@astro.psu.edu

2nd East Asian Workshops in Astrostatistics Summer 2018

Adapted from R scripts in Appendix B, *Modern Statistical Methods for Astronomy With R Applications*, Eric D. Feigelson & G. Jogesh Babu 2012 http://astrostatistics.psu.edu/MSMA (http://astrostatistics.psu.edu/MSMA)

R is written in C and some R functions (particularly vector operations) proceed at machine code speed. But R is an interpreted language, and some functions (e.g. `for`, `if/else` and `while` loops) proceed at much slower speeds.

R functions were changed to a `byte-code compiler` c2012, so on-the-fly compilation is reduced. Python has the same compiler type.

R code can often be improved for performance through improved structure & vectorization, by converting computationally-intensive portions to C or Fortran, by using parallel processing within R, and by using advanced CRAN packages for use on large CPU/GPU clusters or cloud computing.

We now proceed with some tests of operational speed for different coding practices. Advice on speeding up R code can be found in the following references:

- The Art of R Programming, N. Matloff (2011, book, Chpt 11)
- At www.r-bloggers.com: FasteR! HigheR! StrongeR!, N. Ross (2013)
- At www.r-statistics.com: Speed up using JIT compiler, T. Galil (2012)
- Getting Started with doParallel and foreach, S. Weston & R. Calaway (2014)
- Simple Parallel Statistical Computing in R, L. Tierney (slides, 2003)
- State of the Art in Parallel Computing with R, M. Schmidberger et al. (J Stat Software, 2009)
- Tutorial: Parallel Computing with R on Lion and Hammer (RCC/PSU, 2013)
- HPC-R Exercises: Parallelism, D. Schmidt (2015)

I Benchmarking R codes

We find that many R functions, such as the normal and beta distribution random number generators, are vector operations that operate at full speed of the CPU with O(N) scaling. However, a `for` loop is can be 10-100 times slower, and nested `for` loops can be prohibitively time consuming. Note that even simple operators like `:` and <- require function calls that can slow a program.

In [ ]:
```
N <- 1000000 # a million operations

test1 <- function(n) {
        foo1 <- rnorm(n) ; foo2 <- rbeta(n,5,5)
        foo3 <- foo1 + foo2
        return(foo3) }
system.time(test1(N))          # vector operations, fast, R ~ 10*syst
em
system.time(test1(N*10))       # O(N) behavior

test2 <- function(n) {
        foo3 <- vector(length=n)
        foo1 <- rnorm(n) ; foo2 <- rbeta(n,5,5)
        for (i in 1:n) foo3[i] <- foo1[i] + foo2[i]
        return(foo3) }
system.time(test2(N))          # for loop, 10x slower, R ~ 100*system

test3 <- function(n) {
        foo3 <- vector(length=n)
        for (i in 1:n/10)
                for (j in 1:10)
                foo1 <- rnorm(n) ; foo2 <- rbeta(n,5,5)
                        for (i in 1:n) {foo3[i] <- foo1[i] + foo2[i]}
        return(foo3) }
# system.time(test3(10000))    # Double loop, very slow, R ~ 40*syste
m
system.time(test3(3000))       # O(N^2) behavior
```

II Profiling & debugging R programs

Next, we turn to profiling procedures that help identify which steps are slowing the processing of a complicated code. `Rprof` is a utility in base-R while `microbenchmark` is one of several CRAN packages to help with improving the efficiency of R coding. In the case of our `test2` function, we find that most of the time is spent generating random numbers.

In [ ]:
```
Rprof("profile.result")
invisible(test2(N))
Rprof(NULL)
summaryRprof("profile.result")

install.packages('microbenchmark', repos='https://cloud.r-project.org'
)
library(microbenchmark)  ;  library(ggplot2)
compare <- microbenchmark(test1(N/10), test2(N/10), times = 50)
autoplot(compare)
```

R has built-in functions including `debug`, `browser`, `traceback`, `options(error=recover)`, and `tryCatch` to help the programmer understand complex codes. CRAN packages include `debug`. Run R within `gdb` to debug C code called by R scripts.

III Speeding up R

Clever use of the following can often speed up your program: sort, table, inner, outer, crossword, expand.grid, which, where, any, all, sum, cumsum, sumRows, cumprod, %% (modulo), etc.

Following is a slow code with many calls to a random number generator, and a fast code with only one call. This illustrates tradeoff between speed and memory usage. This and other examples of R speedup efforts are given by N. Matloff. A particular problem with R processing is that vectors are often unnecessarily copied and recalculated.

In [ ]:
```
sum_ran2 <- 0
system.time(for (i in 1:N) { ran.2 = rnorm(2) ; sum_ran2 = sum_ran2 +
max(ran.2) } )

system.time (ran.many <- matrix(rnorm(2*N), ncol=2) )
system.time (sum(pmax(ran.many[,1], ran.many[,2])))
```

Many operations can be sped up with R's `apply` functions: apply, sapply, lapply, etc. lapply loops in compiled C code and can be fastest, although using numerics can be faster than using lists. lapply procedures can be parallelized using `mclapply` in package `parallel`. In the example below, the "+" function can be replaced by a more complex user-defined function.

In [ ]:
```
test4 <- function(n) {
        foo1 <- rnorm(n) ; foo2 <- rbeta(n,5,5)
        foo4 <- apply(cbind(foo1, foo2), 1, "+")
        return(foo4)  }
system.time(test4(N))           # apply is not effective here
```

IV Pre-compiling R code

Since c2012, all R/CRAN functions are pre-compiled, but user-defined functions are not. You can do this yourself and speed up your code.

In [ ]:
```
system.time(test2(N))
library(compiler)
comp_test2 <- cmpfun(test2)
system.time(comp_test2(N))      # no improvement here
```

A related option is to use just-in-time (JIT) compiling in R that automatically compiles all functions their first time. Add the following at the beginning of the code.

In [ ]:
```
library(compiler)
enableJIT(1)
```

Conclusion on speeding up R by N. Matloff: "The moral of the story is that performance issues can be unpredictable."

V Parallel processing

Important CRAN packages: multicore, snow,snowfall, doParallel,foreach, and plyr. Most parallelizations are related to `apply`, so if you can run your task in `apply`, you can parallelize.

Easy start to parallel processing: R package `doParallel` as an interface between the `parallel` package, a merger of CRAN's `multicore` and `snow` (Simple Network of Workstations packages, and the `foreach` package/function provided by the company MS Revolution Analytics. This runs both on a single computer with multicores and on a cluster of processors. With `doParallel`, 'an average R programmer can start executing parallel programs, without any previous experience in parallel computing'.

Useful documentation:

- vignette("gettingstartedParallel")
- 'Introduction to parallel computing in R' (Clint Leach, 2014)
- CRAN Task View on High Performance Computing

In [ ]:
```
# Setup multicore cluster
install.packages('doParallel', repos='https://cloud.r-project.org')
library(doParallel)
getDoParWorkers()   # Find number of cores available
clus <- makeCluster(4)
registerDoParallel(clus)
ncores <- getDoParWorkers() ; ncores
stopCluster(clus)
```