Speeding up code in R – 4/23/14

# Measuring system performance

One common way to measure system performance is to measure how long a computer takes to perform a given instruction or set of instructions. The most common way to do this is using system.time(expr), which will return the time a computer took to run expr. However, system.time() is not very precise (measuring in thousandths of a second). If more precision is needed, the function microbenchmark() is able to measure down to nanoseconds. For processes that take longer, reduce the times parameter so that it takes shorter to run (default is 100).

# Making your code faster in three simple steps

1. Find the biggest bottleneck, the slowest part of your code.
2. Eliminate the bottleneck by modifying the code.
3. Repeat until your code is fast enough.

# Finding bottlenecks – profiling with Rprof

The first step to speed up code is to find the bottlenecks in your code. When dealing with large chunks of code, a practical approach is to use system.time() to isolate which parts are taking longer, but once you find those parts you need to understand what is making them slow. Rprof is a convenient tool that tells you how long your code is spending in each of the functions it calls. It works by examining the call stack at fixed interval (default is 0.02 second) to determine which function calls are active then. Then it writes those functions to an output file (specified via filename), which can be summarized using the function summaryRprof(). Rprof can also profile memory use, line locations (which line of your code is taking longer) and garbage collection. Using Rprof is simple:

> Rprof(…) # Start Rprof with options ‘…’

> <whatever\_you\_want\_to\_profile>

> Rprof(NULL) # End Rprof

> summaryRprof() # Summarize results

# Other R implementations that you might want to try

The usual R that you get at CRAN (Gnu-R) is an implementation with more than 20 years of history, kept by the R-Core Team. This is mostly a voluntary effort, so there is still place for a lot of improvement. However, there are alternative implementations of R which improve many aspects of the language (taken from Wickham’s Advanced R):

* **pqR** (pretty quick R), by Radford Neal. It's built on top of the existing R code base (2.15.0), and fixes many obvious performance issues. It provides better memory management, and some support for automatic multithreading. (http://www.pqr-project.org/)
* **Renjin** by BeDataDriven. Renjin uses the java virtual machine, and has an extensive test suite. (http://www.renjin.org/)
* **fastr**, by a team from Purdue. fastr is similar to Renjin, but it makes more ambitious optimisations and is somewhat less mature. (https://github.com/allr/fastr)
* **Riposte**, by Justin Talbot and Zachary DeVito. Riposte is experimental and ambitious, and for the parts of R it implements is extremely fast. Riposte is described in more detail in Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R. (https://github.com/jtalbot/riposte)

# Vectorizing

Vectorized functions are much faster than simple loops or ‘apply’ calls. They are fast because their internal loops are optimized and written in C++, so using them wherever possible is a good way to speed up code.

Here is a list of vectorized functions in R you may not know, and could help speed up your code:

|  |  |
| --- | --- |
| Function | What does it do? |
| ifelse() | Vectorized if. |
| pmax() | Returns the (parallel) maxima. |
| pmin() | Returns the (parallel) minima. |
| rowSums() | Row sums for numeric arrays. |
| colSums() | Column sums for numeric arrays. |
| rowMeans() | Row means for numeric arrays. |
| colMeans() | Column means for numeric arrays. |
| crossprod() | Returns a matrix cross-product of two matrices. |
| which() | Give the TRUE indices of a logical object, allowing for array indices. |
| all() | Are all values TRUE? |
| any() | Is at least one value TRUE? |
| cumsum() | Cumulative sum of a vector. Returns a vector. |
| cumprod() | Cumulative product of a vector. Returns a vector. |
| outer() | The outer product of the arrays X and Y is the array A with dimension c(dim(X), dim(Y)) where element A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...). |
| lower.tri() | Returns a matrix of logicals the same size of a given matrix with entries TRUE in the lower triangle. |
| upper.tri() | Returns a matrix of logicals the same size of a given matrix with entries TRUE in the upper triangle. |
| expand.grid() | Create a data frame from all combinations of the supplied vectors or factors. |

# Byte code compilation

Some types of R code can be byte code compiled, which can increase their speed. It is very easy to use and can work for many functions, but in most cases the speedups are going to be less than spectacular. Note that all the base R functions are already byte code compiled by default. To use it, you need package compiler (already in the base R). There are several useful functions in the package, for example, cmpfun compiles the body of a closure and returns a new compiled closure with the same formals; compile compiles an expression into an object which can then be evaluated using eval; cmpfile compiles a file, which can later be sourced using loadcmp.

# Parallelizing code in R

R is by default single-threaded. To take advantage of multiple, you will need to use one of the many packages developed for this purpose. Here, we will be looking at three of the most popular approaches, but for a comprehensive list of what is available, see:

http://cran.r-project.org/web/views/HighPerformanceComputing.html

## Using snow and snowfall

snow (**s**imple **n**etwork **o**f **w**orkstations) is a package developed by Luke Tierney and others intended to easily implement parallel computing for embarrassingly parallel problems. snowfall, developed by Jochen Knauss and others, is a package intended to further simplify the use of snow by automating the creation of clusters and assignment of tasks.

Before parallelizing, you need to write the code you want to parallelize inside a wrapper function. The first argument of this function needs to be an iterator over which you’re going to parallelize. Once you have this in place, using the packages is relatively easy:

> fun <- function(ii, …){}

> require(“snow”)

> require(“snowfall”)

First, initialize a cluster. You can tell it whether to run in parallel or serially (parallel), what type of cluster to create (type, options include SOCK, MPI, PVM and NWS; use SOCK for a multicore single machine), number of cpus to use (cpus), among other options.

> sfInit(parallel=TRUE, type=’SOCK’, cpus=4)

If you need to pass any objects, libraries or source codes to each one of the workers, you can use the functions sfExport(), sfLibrary(), sfSource() and sfExportAll(). Then, to parallelize, you can use the functions sfApply(), sfLapply() and sfSapply(), which are the parallel versions of apply(), lapply() and sapply().

> sfSapply(ii, fun, …)

And remember to stop the cluster once you’re done.

> sfStop()

For more information, see:

http://journal.r-project.org/archive/2009-1/RJournal\_2009-1\_Knaus+et+al.pdf

## Using parallel

The parallel package is based on packages multicore and snow, and provides drop-in replacements for most of the functionality of those packages, with integrated handling of random-number generation. It also uses a lapply-like interface, so the first step will be to create a wrapper function for what you want to parallelize.

> fun <- function(ii, …){}

> require(“parallel”)

In Windows, you will need to initialize a cluster, telling it what type of cluster and how many workers to use

> numWorkers <- 4

> cl <- makeCluster(numWorkers, type = “PSOCK”)

Then, run the function to parallelize using parLapply()

> parLapply(cl, ii, fun, …)

And as before, stop the cluster

> stopCluster(cl)

If you’re running on a Mac or Linux, you can skip the setting up clusters steps and instead use mclapply(), which works by forking workers (not supported by Windows).

> mclapply(ii, fun, mc.cores=numWorkers, …)

For more information, see: http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf

## Using foreach

The foreach package, developed by Steve Weston from Revolution Analytics, differs from the previous two packages in that it uses a for-loop construction. To use foreach in parallel, you will also need the package doParallel. As with the parallel package, start by initializing a cluster, and the register it for use with foreach

> numWorkers <- 4

> cl <- makeCluster(numWorkers, type = “PSOCK”)

> registerDoParallel(cl)

With this done, you can now run your code in parallel as you would do with a for-loop

> foreach(ii = <some\_sequence>, …) %dopar% { }

The binary operator %dopar% tells foreach to do this in parallel. If you need to do it serially, use %do%. Other options that can be passed to foreach are .combine (how to process the task results as they are generated), .init (initial value for .combine), .inorder (tells .combine that tasks need to be combined in order), .packages (packages a task depend on), .export (variables to export). Once you’re done, don’t forget to stop the cluster

> stopCluster(cl=NULL)

For more information, see:

http://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf

http://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf