Scalable Programming Strategies for Massive Data in R

Jay Emerson

April 6, 2017

Why? Who?

- Distance or adjacency matrices (talk #1 by Nicolas)
- Lots of large images (talk #2 by Hugo)
- Some Bayesian computations (talk #3 by Rémi)
- ...

- Potential "high-level" work in R without any C/C++ coding (i.e. you don't need to be an expert, but need to be thoughtful)
- Potential "low-level" code development (for experts or ambitious advanced/intermediate R programmers)

Abstract

Computing with native R objects is limited to available memory (RAM), lacks shared-memory capabilities, and frequently incurs costly memory overhead. This talk will present three specific examples with take-away material: (1) using package *bigmemory* for storing and interacting with massive matrices (2) using package *foreach* as an elegant and portable framework for parallel computing, and (3) building a basic R package that includes C/C++ code for an algorithm that scales beyond the constraints of RAM.

The Plan

- Simple toy example illustrate the important points
- I'll try to focus on the "big picture" – providing starting points for further work/exploration
- I hope you find something here ***useful*** – "high-level" and/or "low-level"

- Today:
  - I'll start with slide 28 of 33 to make sure I finish
  - I'll move into R to run some examples interactively
  - I'll return to this slide deck for some more introductory examples and recommended methods for parallel computing; these examples stand on their own if we run out of time.
  - I'll conclude with a peek at the most important "take-away material"

Toy Examples: From Memory Overhead to Parallel Programming

Memory overhead: setup

```
gc(reset=TRUE)
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 372445 19.9     592000 31.7   372445 19.9
## Vcells 574838  4.4    1308461 10.0   574838  4.4
```

```
x <- rep(0L, 1000000); is.integer(x)
```

```
## [1] TRUE
```

```
object.size(x)
```

```
## 4000040 bytes
```

Memory Overhead: surprising?

```
x <- x + 1
is.integer(x); is.integer(1)
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
object.size(x)
```

```
## 8000040 bytes
```

```
gc()
```

```
##            used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  373573 20.0     750400 40.1   433202 23.2
## Vcells 1575926 12.1    2759800 21.1  2116501 16.2
```

Memory Overhead: read.csv() example

1995.csv is a 500 MB CSV file and produces a 700 MB data.frame:

```
> system("ls -als | grep 1995.csv")
... (edited) ...          530751568 Mar 24  2010 1995.csv
> gc(reset=TRUE)
... 12.2 MB      # the baseline memory usage
> x <- read.csv("1995.csv", as.is=TRUE)
> gc()
... 2672.2 MB    # peak memory usage during read.csv()
> object.size(x)
703415096 bytes
```

Memory Overhead: lm() and others...

I'm going to skip these examples, but here's a general recommendation when working with standard R objects: any one object should be not more than 10-20% of available RAM and the total objects in the environment shouldn't be more than about 1/3 of total RAM.

Here's a somewhat-related thought that I'll come back to, later, from the *R Installation and Administration* guide of the R Project:

```
Even on 64-bit builds of R there are limits on the size
of R objects (see help("Memory-limits")), some of which
stem from the use of 32-bit integers (especially in
FORTRAN code). For example, the dimensions of an array
are limited to 2^{31} - 1.
```

Speed Baseline

I'll use this example task with a wide range of solutions: find the 250,000 column sums of a matrix that has 4 rows and 250,000 columns. The use of `apply()` to do this is very common and will thus be our benchmark:

```
# `apply`
x <- matrix(rnorm(1e6), nrow=4) # 4 x 250,000
system.time({
  ans <- apply(x, 2, sum)
})
```

```
##    user  system elapsed
##   0.434   0.012   0.445
```

A bad for loop

```
# A bad `for` loop: Why?
system.time({
  ans <- NULL
  for (i in 1:ncol(x)) {
    ans <- c(ans, sum(x[,i]))
  }
})
```

```
##    user  system elapsed
##  92.853  15.486 108.544
```

A better for loop

```
# A better `for` loop
system.time({
  ans <- rep(0, ncol(x))
  for (i in 1:ncol(x)) {
    ans[i] <- sum(x[,i])
  }
})
```

```
##    user  system elapsed
##   0.482   0.003   0.486
```

Rcpp first example

I'll provide references in take-away material. If you're going to mess around in the sandbox with R and C/C++, you definitely want to be using *Rcpp*.

```
# `Rcpp` (requires compilers and tools)
library(Rcpp)
```

The following page is an R function call whose argument is a character string that just happens to contain text that is C++ code. Of course R is unaware of this, but `sourceCPP()` handles the magic.

Rcpp first example

```
sourceCpp(code='
            #include <Rcpp.h>
            using namespace Rcpp;
            // [[Rcpp::export]]
            NumericVector mycolsum(NumericMatrix x) {
            NumericVector ans(x.ncol());
            int i, j;
            for (j=0; j<x.ncol(); j++) {
              ans[j] = 0;
              for (i=0; i<x.nrow(); i++) {
                ans[j] += x(i,j);
              }
            }
            return ans;
            }')
```

Rcpp in action

We'll come back to *Rcpp* later...

```
system.time({
  ans <- mycolsum(x)
})
```

```
##    user  system elapsed
##   0.006   0.000   0.007
```

Package parallel (included with R)

This package basically provides simple wrappers to *SNOW* and *multicore*.

- *SNOW*
  - works on all platforms
  - can work with clusters
  - unfortunately, makes copies of necessary objects for each worker
- *multicore*
  - not supported in Windows
  - works on SMP hardware (single computers/workstations)
  - is more memory efficient than *SNOW* as long as the use is read-only

```
library(parallel)  # Comes automatically with R now
```

parallel: an example via SNOW

```
cl <- makeCluster(c("localhost","localhost"))
parSapply(cl, 1:9, function(x) x^2)
```

```
## [1]  1  4  9 16 25 36 49 64 81
```

```
stopCluster(cl)
```

parallel: an example via multicore

```
unlist( mclapply(1:9, function(x) x^2) )
```

```
## [1]  1  4  9 16 25 36 49 64 81
```

foreach

- an alternative (no longer "new") looping construct for R that supports parallel computing
- abstracts away the parallel communication infrastructure
  - the user can chose between *SNOW*, *multicore*, *Rmpi*, ... via the so-called "parallel backend" packages *doSNOW*, *doMC*, *doMPI*, *doRedis*, ...
  - the "interesting work" is portable and (mostly) independent of the choice of the parallel backend
  - an ideal choice for *supporting* high-level parallelism in your package but not *forcing* it
- has some very elegant design features
- works with iterators (to be discussed briefly)

foreach with doMC

```
# Loading doMC also loads foreach:
suppressMessages(library(doMC))

# Declare the number of processor cores and go for it:
registerDoMC(2)
foreach(i = 1:9, .combine=c) %dopar% {
  return(i^2)
}
```

```
## [1]  1  4  9 16 25 36 49 64 81
```

foreach with doSNOW: setup (slide-friendly)

```
# Loading doSNOW also loads foreach:
suppressMessages(library(doSNOW))

# Now declare the number of processor cores and
# there are many more options available, including
# support for parallel computing on a cluster.

cl <- makeCluster(2, type="SOCK")
registerDoSNOW(cl)
```

The real work is on the next slide...

foreach with doSNOW: usage, stoping cluster

```
# The real work:
foreach(i = 1:9, .combine=c) %dopar% {
  return(i^2)
}
```

```
## [1]  1  4  9 16 25 36 49 64 81
```

```
stopCluster(cl) # Don't forget this!  Saves trouble...
```

A really bad attempt at parallel programming

```
dim(x)
```

```
## [1]      4 250000
```

```
registerDoMC(2) # 2 processor cores
system.time({
  ans <- foreach(i=1:ncol(x),
                 .combine=c) %dopar%
                 {
                   return(sum(x[,i]))
                 }
})
```

```
##    user  system elapsed
##  68.019   1.395  69.417
```

A much better approach (and iterators are cool)

```
library(itertools)
registerDoMC(2)
system.time({
  iter <- isplitIndices(ncol(x), chunks=4)
  ans <- foreach(i=iter,
                 .combine=c) %dopar%
                 {
                   return(apply(x[,i], 2, sum))
                 }
})
```

```
##    user  system elapsed
##   0.012   0.014   0.356
```

A basic iterator (slide 1)

```
iter <- isplitIndices(11, chunks=3)
iter
```

```
## $nextElem
## function ()
## {
##     m <- as.integer(nextElem(it))
##     j <- i
##     i <<- i + m
##     seq(j, length = m)
## }
## <environment: 0x7f9a00c96cb0>
##
## attr(,"class")
## [1] "abstractiter" "iter"
```

The basic iterator (slide 2)

```
nextElem(iter)
```

```
## [1] 1 2 3 4
```

```
nextElem(iter)
```

```
## [1] 5 6 7 8
```

```
nextElem(iter)
```

```
## [1]  9 10 11
```

And via colSums()... and why this misses the point...

Look, if all you need to do is take column sums – fine. What you're probably interested in, though, is a framework for developing/implementing new and useful methods. The work is more complicated than what we're doing here, but the foundation remains the same.

```
system.time({
  ans <- colSums(x)
})
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

Package *bigmemory* and friends (*bigtabulate*, *biganalytics*, *bigalgebra*, *synchronicity*, ... and a growing list of packages by other authors that use *bigmemory*)

A big matrix (object of type big.matrix)

Create a 500,000,000-row, 8-column matrix of doubles (about 4 GB per column so 32 GB total, well in excess of my RAM):

```
x <- big.matrix(nrow=5e8,
                ncol=8,
                init=0, # Optional but a good idea
                backingfile="big.bin",
                descriptorfile="big.desc")
```

Be careful. A `big.matrix` is not a `matrix` and definitely isn't a `data.frame`. It can't be used with most R functions and packages. But it can be amazing for helping manage and work with massive data... with proper care taken.

The Result (mildly edited...)

```
JWEs-MacBook-Pro:Scratch jay$ ls -als
total 62500024
... jay  staff          170 Mar 24 13:33 .
... jay  staff         1020 Mar 25 10:13 ..
... jay  staff  32000000001 Mar 24 13:51 big.bin
... jay  staff          463 Mar 24 13:33 big.desc

JWEs-MacBook-Pro:Scratch jay$ more big.desc
new("big.matrix.descriptor",
    description = structure(list(sharedType = "FileBacked",
    totalRows = 500000000L, totalCols = 8L, rowOffset = c(0
    5e+08), colOffset = c(0, 8), nrow = 5e+08, ncol = 8, ro
    colNames = NULL, type = "double", separated = FALSE),
    "filename", "totalRows", "totalCols", "rowOffset", "col
    "nrow", "ncol", "rowNames", "colNames", "type", "separa
)
```

More examples

Skip over to an interactive R session now: Demo.R.
This doesn't go well into slides.

Scalable R Package Infrastructure: Authoring R Packages with *Rcpp* and *bigmemory*

Materials

`http://www.stat.yale.edu/~jay/StatLearn2017/`

(locally, RPC/ in Travel/Lyon/EmersonTalk folder)

Me: Jay (john.emerson@yale.edu)