R with High Performance Computing: Parallel processing and large memory

Amy F. Szczepański, Remote Data Analysis and Visualization Center, University of Tennessee
`http://rdav.nics.tennessee.edu/`

Many thanks to Pragnesh Patel for all of his help and for providing so much nice example code.

1. Today we will cover:
   - NSF's HPC resources
   - A bit about the architecture of Nautilus.
   - A brief overview of how to run R via a batch system.
   - A bit about timing and profiling.
   - Parallelizing without rewriting.
   - Parallelizing with minor rewriting.
   - Some words about R with MPI or clusters.
   - Some words about R with GPUs.
   - R with large memory.
2. NSF's Office of Cyber Infrastructure (OCI), through the Extreme Science and Engineering Discovery Environment (XSEDE), operates high performance computing resources for researchers doing open science. Scientists can apply for time on these systems via `http://portal.xsede.org`. The computational resources range from small clusters to very large systems like Ranger (Texas Advanced Computing Center, 62,976 cores) and Kraken (National Institute for Computational Sciences, 112,896 cores). There are also systems for visualization and data analysis, such as Nautilus (Remote Data Analysis and Visualization Center, 1024 cores and 4 TB shared memory) and Longhorn (TACC, 2048 compute cores, 512 GPUs, and 13.5 TB distributed memory). XSEDE also has resources for data storage, data movement, science gateways, and other tools to facilitate computational science.
3. Shameless plug: If your work could be construed as "open science" research (projects being pursued with the intent to publish in the scientific literature), consider using Nautilus for your data analysis and visualization. Please get in touch (`aszczepa@utk.edu`) if you are interested! If you want to explore on your own, create an account at `http://portal.xsede.org` and click on **allocations**. Our system is referred to on the allocation page as "NICS SGI/NVIDIA, Visualization and Data Analysis System (Nautilus)." We want to help you get science done!
4. Shameless plug #2: Find the **Remote Data Analysis and Visualization Center** on Facebook and follow **@NICS_Nautilus** on Twitter.

   Check us out with:

   ```
   install.packages(twitteR)
   library(twitteR)
   NautilusTweets <- userTimeline("NICS_Nautilus")
   NautilusTweets
   ```

   Following us from R is a bit more complicated—it requires registering your twitteR connection with OAuth.
5. Since Nautilus is an SMP, many of these examples would run much the same way on Nautilus as they would on your own laptop. I'll likely be running most of these examples on my own computer

(depending on the queue situation on Nautilus and the availability of Internet access when we hold this session).

6. Most work on HPC systems is done via scripts submitted at the command line. Furthermore, we submit these jobs through a batch environment. On Nautilus we use Moab with TORQUE.

7. Here is an example of a typical PBS script for Nautilus. This script assumes that our R script is in our home directory and we issue the command from our home directory.

```
#!/bin/bash

#PBS -N myJobName
#PBS -q analysis
#PBS -j oe
#PBS -l ncpus=16
#PBS -l mem=64gb
#PBS -l walltime=1:00:00

module load r
Rscript myscript.R
ja
```

   It asks for 16 CPUs and 64 GB of RAM for an hour in the analysis queue. On Nautilus, jobs in the analysis queue can pre-empt jobs in the computation queue.

   We would save this script into a file named **mysubmission.pbs** and submit it to the queue with the command `qsub mysubmission.pbs`. The system would then place your job in the queue and then run it once it reaches the front of the line. Once it runs, it would save all of its output in a file named by the job name. This script is assuming that everything happens from the user's home directory.

   In some of our scripts we'll need to set environment variables to control threaded behavior. We'll typically insert those lines after the PBS options but before we give the command `module load r`.

   The `ja` command at the end of the script tells the system to report some **j**ob **a**ccounting in the output file.

8. To compare the performance of parallel and serial calculations, we will need to quantify performance. The easiest way to find out how long it takes an R function to run is with the function `system.time()`. For example, we could say `system.time(2*3)`. We can also use `proc.time()` to find out how long our current process has taken.

9. In addition to using `system.time()` and `proc.time()` we can use `Rprof` to profile our code. Here is some code that creates a matrix X filled with random numbers from a normal distribution and calculates $X^t X$ three ways:

   (a) Element by element, using the rule that we use "by hand" with %o% for the dot product.

   (b) Using the matrix multiplication operation %*% to multiply X^t by X.

   (c) Using the statistics function `crossprod()`.

   Then it calls the `all.equal()` function to make sure that it got the same answer from all three calculations.

```
its = 2500
dim = 1750
X = matrix(rnorm(its*dim),its, dim)
```

```
my.cross.prod <- function(X)
{
    C = matrix(0, ncol(X), ncol(X))
    for(i in 1:nrow(X))
    {
        C = C + X[i,] %o% X[i,]
    }
    return(C)
}

Rprof("matrix-mult.out")
C  = my.cross.prod(X)

C1 = t(X) %*% X

C2 = crossprod(X)
Rprof(NULL)

print(all.equal(C,C1,C2))
quit(save="no")
```

After we run this code, we have a fille called **matrix-mult.out** that has the timing information in it. We would use `R CMD Rprof matrix-mult.out` to see which functions we spend the most time in.

10. There are other ways for us to profile out code. We can also use the proftools package.

```
library(proftools)
profile.data <- readProfileData("matrix-mult.out")
flatProfile(profile.data)
```

We get an output like:

```
               total.pct total.time self.pct self.time
my.cross.prod      83.13      80.74     0.16      0.16
%*%                55.11      53.52    55.11     53.52
%o%                45.20      43.90     0.00      0.00
outer              45.20      43.90     1.07      1.04
+                  37.77      36.68    37.77     36.68
crossprod           5.44       5.28     5.44      5.28
t                   0.31       0.30     0.16      0.16
t.default           0.14       0.14     0.14      0.14
as.vector           0.10       0.10     0.10      0.10
==                  0.02       0.02     0.02      0.02
vector              0.02       0.02     0.02      0.02
```

11. Speed-ups for the clever: Depending what you need, you might choose to use faster functions. Here is an example that compares `lm()` with `lm.fit()`. This uses the **diamonds** data set that comes with ggplot2. It fits a linear model of the **price** of the diamond as a function of its **carat** weight. The `lm()` function calculates everything we want to know about the linear model; the `lm.fit()` function calculates less information (and requires that you pass it a design matrix as an input). This example is included for two reasons: both as an example of a faster function and also a nice example for graphical profiling with the profr package.

```
library(ggplot2)
data(diamonds)

Rprof("model1.out")
my.model1 <- lm(diamonds$price ~ diamonds$carat)
Rprof(NULL)

my.matrix <- data.matrix(data.frame(intercept=1, diamonds$carat))

Rprof("model2.out")
my.model2 <- lm.fit(x=my.matrix, y=diamonds$price)
Rprof(NULL)

print(my.model1)
print(my.model2$coefficients)
```

We can compare these with `R CMD Rprof` run from a shell on our computer (not from within R).

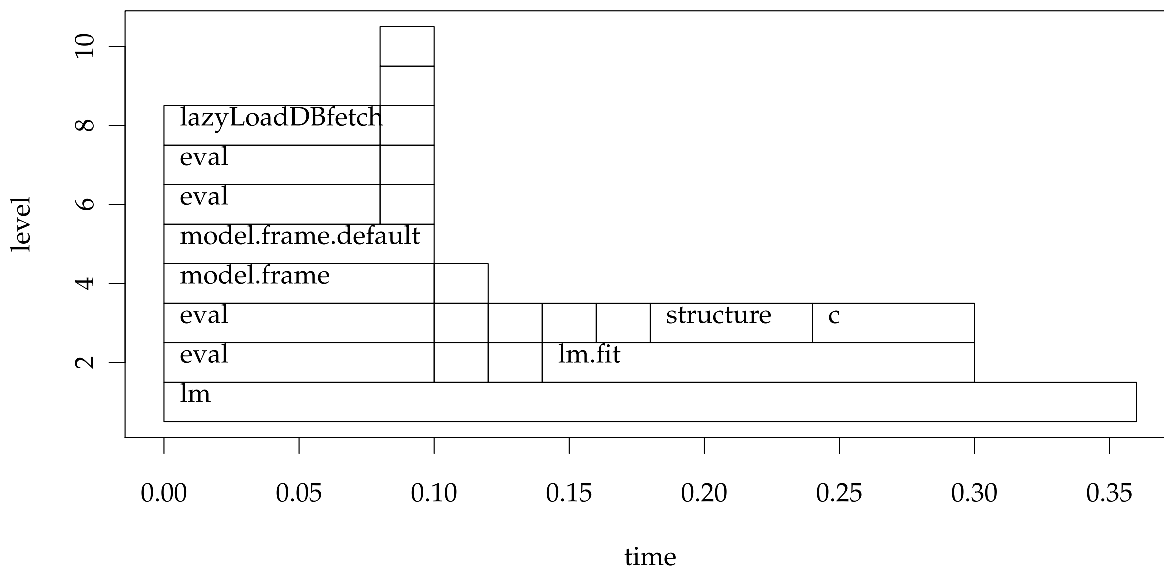
```
R CMD Rprof model1.out
R CMD Rprof model2.out
```

During my run, I found that `lm()` took 0.36 seconds to run and `lm.fit()` took 0.06 seconds to run.

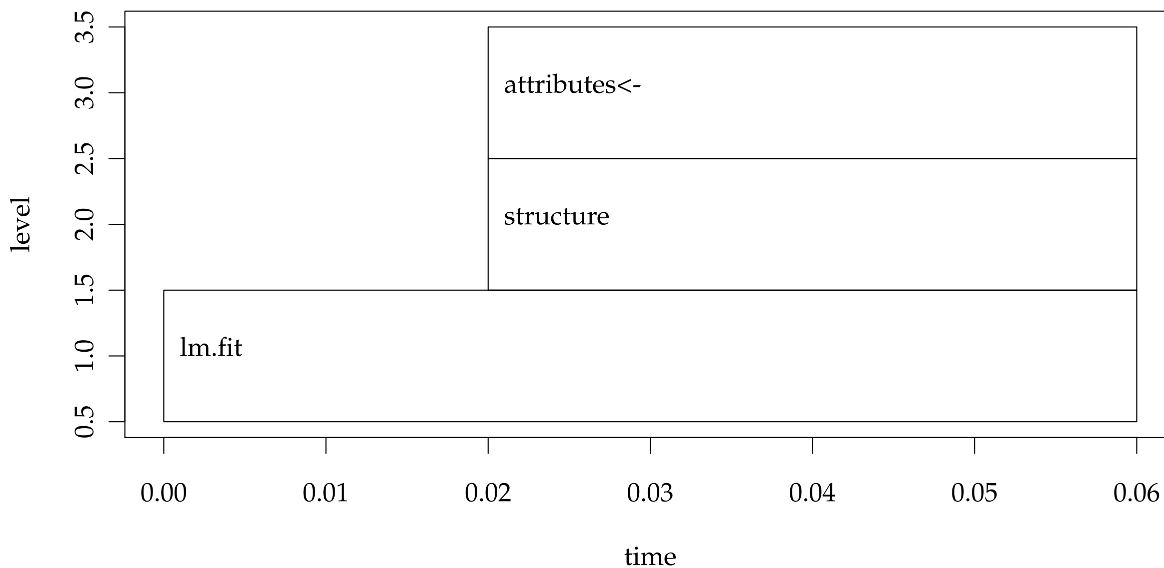
12. We can also use the following R script (with the `quartz()` command for running interactively from within R on a Macintosh) to visualize our results:

```
library(profr)
plot(parse_rprof("model1.out"))
quartz()
plot(parse_rprof("model2.out"))
```

Profile of the `lm()` function



Profile of the `lm.fit()` function



13. Speed-ups for free: If our installation of R is compiled with the Intel MKL library, we'll automatically get multi-threaded implementations of standard linear algebra functions. Here is an example of an R script that demonstrates this. This script will do the same calculation as above, timing each method:

```
its = 2500
dim = 1750
X = matrix(rnorm(its*dim),its, dim)

# single thread breakup calculation
system.time({C=matrix(0, dim, dim);for(i in 1:its)C = C + (X[i,] %o% X[i,])})

# BLAS matrix mult
system.time({C1 = t(X) %*% X})

# BLAS matrix mult
system.time({C2 = crossprod(X)})

print(all.equal(C,C1,C2))
quit(save="no")
```

To run this on eight cores on Nautilus, we set the following environment variables in our PBS script:

```
export MKL_NUM_THREADS=8
export MKL_DYNAMIC=FALSE
```

And here is the output that we get:

```
   user  system elapsed
 74.388   6.488  81.328
   user  system elapsed
  2.096   0.020   2.116
```

```
   user  system elapsed
  1.096   0.004   1.100
[1] TRUE
```

14. Another easy way to get a speed-up is to use the pnmath package in R. This package takes many of the standard math functions in R and replaces them with multi-threaded versions, using OpenMP. Here's a quick example of some code that you could run in an interactive R session to try this out. It creates two vectors of random numbers (uniform distribution) and applies some math functions to them. Some functions get more of a speed-up than others with pnmath.

```
v1 <- runif(1000)
v2 <- runif(100000000)

system.time(qtukey(v1,2,3))
system.time(exp(v2))
system.time(sqrt(v2))

library(pnmath)
system.time(qtukey(v1,2,3))
system.time(exp(v2))
system.time(sqrt(v2))
```

15. Here are some timings of some R functions using pnmath on 4 and 8 cores on Nautilus:

Function	**Four cores**	**Eight cores**
`sqrt()`	0.207	0.199
`exp()`	0.069	0
`dnorm()`	0.055	0.001
`lgamma()`	0.013	0.013
`dpois()`	0	0
`df()`	0.019	0.009
`pt()`	0.004	0.002
`qchisq()`	0.04	0.02
`psigamma()`	0.007	0.004
`qbeta()`	0.062	0.031
`qnchisq()`	5.356	2.683
`ptukey()`	61.436	30.602
`qtukey()`	334.185	165.328

16. When using pnmath, we can set the number of threads in our R script with `setNumPnmathThreads(8)` or we can set the number of threads in our environment variables in our PBS script with `export OMP_NUM_THREADS=8`. The default version of this package was written to run on at most eight cores; running on more than eight cores requires some tweaks to pnmath.

17. Some methods of running R in parallel require us to write our code in a certain way. We'll look at loop parallelism next. Once by applying a function to every item in a list and then at parallel `for()` loops.

18. The multicore package has a function `mclapply()` that allows us to apply a function to every item in a list (**m**ulti**c**ore **l**ist **apply**). It also has the function `pvec()` that allows us to apply a function to every element of a vector. We load this package with `library(multicore)`. We can use the following commands to check how many cores are available, to set the number of cores to use, or to see how many cores we are using:

```
library(multicore)
multicore:::detectCores()
options(cores = 8)
getOption('cores')
```

Note that multicore relies on `fork()` and spawns new processes.

19. A simple example with `mclapply()` or `pvec()`:

```
x <- mclapply(1:1000, sqrt)
y <- pvec(1:1000, sqrt)
```

The main difference between these two has to do with assumptions on how the calculation can be applied to the vector.

20. A slightly more complicated example, comparing `mclapply()` to `lapply()`:

```
test <- lapply(1:10, function(x) rnorm(100000))
system.time(x <- lapply(test, function(x) loess.smooth(x,x)))
system.time(x <- mclapply(test, function(x) loess.smooth(x,x)))
```

21. This package also has a `parallel()` and `collect()` construct.

22. The next level of complication is the foreach package from Revolution Analytics. With this package, you can have a `for()` loop run in parallel. It requires you to tell R which method of parallelization you want to use. There are many options, including multicore (which we just saw), SMP, Rmpi, snow, and others. You tell R that you want the loop to run in parallel by specifying `%dopar%`. For it to run in serial, you would say `%do%`.

```
library(multicore)
library(doMC)
library(foreach)

registerDoMC()

foreach(i=1:10) %dopar%
{
    test <- rnorm(100000)
    loess.smooth(test, test)
}
```

23. Other ways of parallelizing code in R include snow (**s**imple **n**etwork **o**f **w**orkstations) and Rmpi. With snow, you define a **cluster** and then can use functions such as `clusterApply()` to have each node in a cluster apply a function to an item from a list. Rmpi is pretty much what it sounds like. For some clusters, a combination of spreading loop iterations out to the nodes and using pnmath so that each node runs multithreaded math operations can be fairly effective.

24. There is a project in the works to connect R to OpenMP functions written in other languages, such as Fortran. The Fortran code is compiled on the fly and imported as a shared object into R.

25. The gputools package provides R interfaces to handful common statistical algorithms. They are implemented using mixture of CUDA language, CUBLAS library and CULA library. It contains many other functions, including hierarchical clustering, SVM training, SVD, Least-squares fit, linear modeling, and many others. Less-communicative algorithms seeing speedups over 20 times on data set of moderate size, but speed up factors vary with CPU, memory configurations and, of course, GPU. Here is an example:

```
library(gputools)
matA <- matrix(runif(3*2), 3, 2)
matB <- matrix(runif(3*4), 3, 4)
gpuCrossprod(matA, matB) # Perform Matrix Cross-product with a GPU

numVectors <- 5
dimension <- 10
Vectors <- matrix(runif(numVectors*dimension), >numVectors, dimension)
gpuDist(Vectors, "euclidean")
gpuDist(Vectors, "maximum")
gpuDist(Vectors, "manhattan")
gpuDist(Vectors, "minkowski", 4)
```

26. Although Nautilus has 4 TB of memory and we've installed a 64-bit version of R, there is a problem: The indexing is limited to 32-bit integers. Consider the packages bigmemory, biganalytics, bigalgebra, and bigtabulate when you want to work with large datasets. The data structures may be allocated to shared memory, allowing separate processes on the same computer share access to single copy of the date set. The data structures may also be file-backend allowing users to easily manage and analyze data sets larger than available RAM and share them across nodes of a cluster.

27. Summary of these packages:

    bigmemory: supports the creation, manipulation and storage of large matrices.

    bigalgebra: provides linear algebra functionality with large matrices.

    biganalytics: extends the functionality of bigmemory.

    bigtabulate: supports `table()`, `split()` and `tapply()` like functionality for large matrices.

    foreach + bigmemory: a winning combination for massive data concurrent programming.

28. Here is an example that uses a very, very large matrix. This example illustrates how to work with a matrix with more than $2^{31} - 1$ elements.

```
# big.matrix: no 2^31-1 object size limitation.

library(bigmemory)
R <- 3e9 # 3 billion rows
C <- 2 # 2 columns

print("48 GB total size:")
R*C*8 # 48 GB total size

date()

x <- filebacked.big.matrix(R, C, type='double',backingfile='huge-data.bin',
descriptorfile='huge-data.desc')
## Generates huge-data.bin and huge-data.desc files.
## Now we can use huge-data.desc file in any R session.

x[1,] <- rnorm(C)
x[nrow(x),] <- runif(C)
summary(x[1,])

summary(x[nrow(x),])
```

```
date()
```

Note: This example *will* leave a 48 GB file on your hard drive!

29. Here is another series of examples that work with a large data set. This uses the airline data set. You can download the data from `http://stat-computing.org/dataexpo/2009/the-data.html`. This example is based on the year 2008 data, which comes in the file **2008.csv.bz2** and can be extracted with `bzip2 -d *.bz2`, giving you the 659 MB spreadsheet.

    This script will create a binary file-backing for this matrix, which we name **airline.bin** and a descriptor file that we name **airline.desc**. These files allow R to access the large data set more quickly.

```
library(bigmemory)
library(biganalytics)

x <- read.big.matrix("2008.csv", type="integer", header=TRUE,
  backingfile="airline.bin", descriptorfile="airline.desc", extraCols="Age")

summary(x)
```

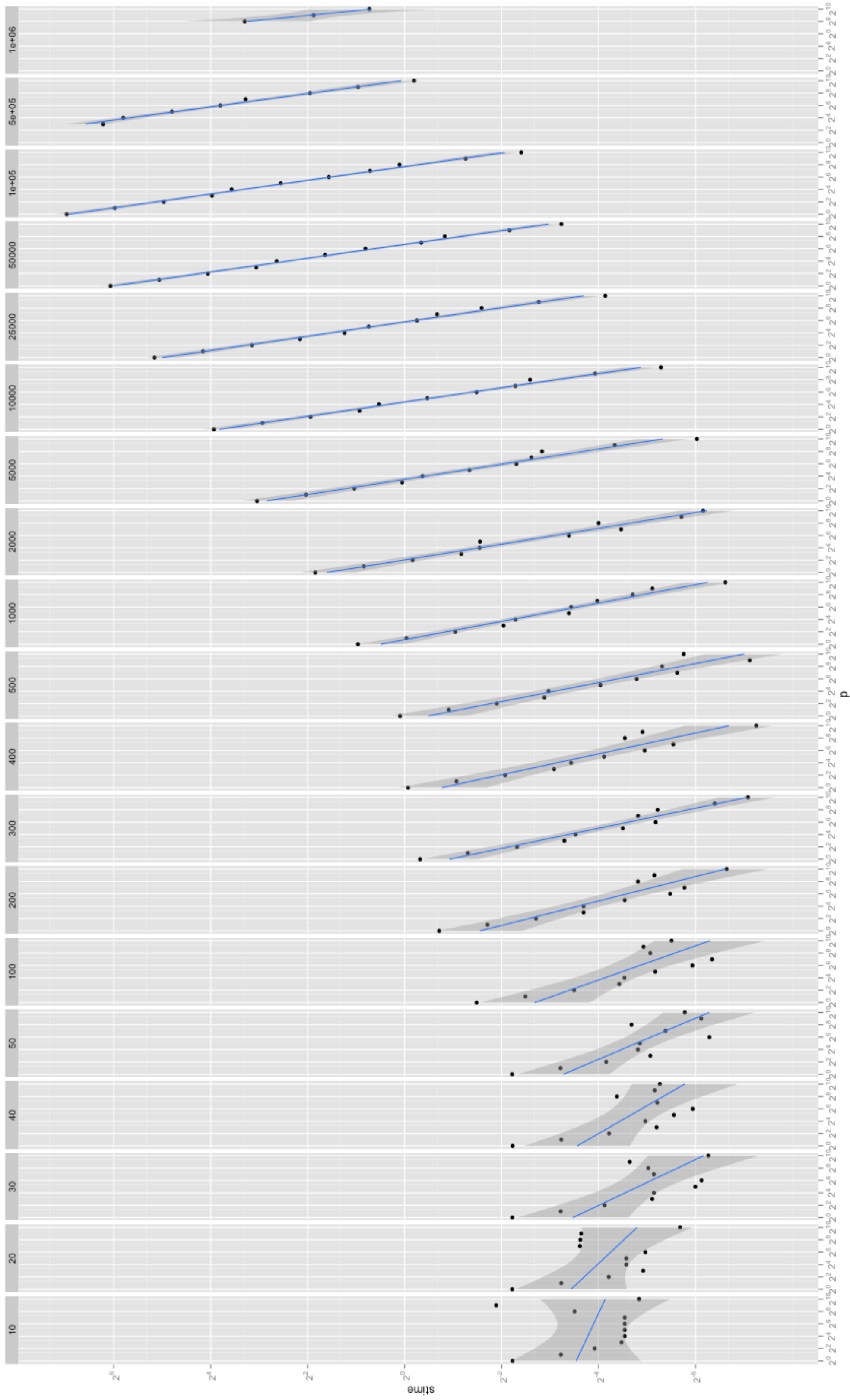
30. This next script will let us use this backing file to access the data quickly in later R sessions, without having to take several minutes to read the spreadsheet.

```
library(bigmemory)
library(biganalytics)

xdesc<- dget("airline.desc") ## we do not need to read all data from csv again.
x<-attach.big.matrix(xdesc)

system.time(numplanes<-colmax(x,"TailNum", na.rm=TRUE))
system.time(numplanes<-colmax(x,"TailNum", na.rm=TRUE))

system.time(colmin(x, 1))
system.time(a <- x[,1])
system.time(a <- x[,2])

colnames(x)
tail(x, 1)
```

A note on the typesetting and graphics: This document was prepared in LaTeX, and the R graphics in the profiling section were output to TikZ files with the tikzDevice package in R and rendered with the tikz package in LaTeX. They were rendered with the following code:

```
library(tikzDevice)
tikz(file="model1.tex", height=3.75, width=6.5)
plot(parse_rprof("model1.out"), main="Profile of the \\texttt{lm()} function")
dev.off()
tikz(file="model2.tex", height=3.75, width=6.5)
plot(parse_rprof("model2.out"), main="Profile of the \\texttt{lm.fit()} function")
dev.off()
```

In this document I added `\usepackage{tikz}` to the preamble and inserted the graphics with `\include{"model1.tex"}` and `\include{"model2.tex"}`.



stime
p
10
20
30
40
50
100
200
300
400
500
1000
2000
5000
10000
25000
50000
1e+05
5e+05
1e+06