Scalable Strategies for Computing with Massive Data: The Bigmemory Project

John W. Emerson and Michael J. Kane
http://www.bigmemory.org/

http://www.stat.yale.edu/~jay/
Associate Professor of Statistics, Yale University
(Professor Emerson prefers to be called "Jay")

Mike submitted his dissertation on August 31, 2010,
and will be Research Faculty, Yale Center for Analytical Science

Please feel free to ask questions along the way!
http://www.stat.yale.edu/~jay/Brazil/SP/bigmemory/

Outline

1. Motivation and Overview
2. Strategies for computing with massive data
3. Modeling with Massive Data
4. Conclusion

A new era

The analysis of very large data sets has recently become an active area of research in statistics and machine learning. Many new computational challenges arise when managing, exploring, and analyzing these data sets, challenges that effectively put the data beyond the reach of researchers who lack specialized software development skills of expensive hardware.

- "We have entered an era of massive scientific data collection, with a demand for answers to large-scale inference problems that lie beyond the scope of classical statistics." – Efron (2005)
- "classical statistics" should include "mainstream computational statistics." – Kane, Emerson, and Weston (in preparation, in reference to Efron's quote)

Example data sets

- Airline on-time data
  - 2009 JSM Data Expo (thanks, Hadley!)
  - About 120 million commercial US airline flights over 20 years
  - 29 variables, integer-valued or categorical (recoded as integer)
  - About 12 gigabytes (GB)
  - `http://stat-computing.org/dataexpo/2009/`

- Netflix data
  - About 100 million ratings from 500,000 customers for 17,000 movies
  - About 2 GB stored as integers
  - No statisticians on the winning team; hard to find statisticians on the leaderboard
  - Top teams: access to expensive hardware; professional computer science and programming expertise
  - `http://www.netflixprize.com/`

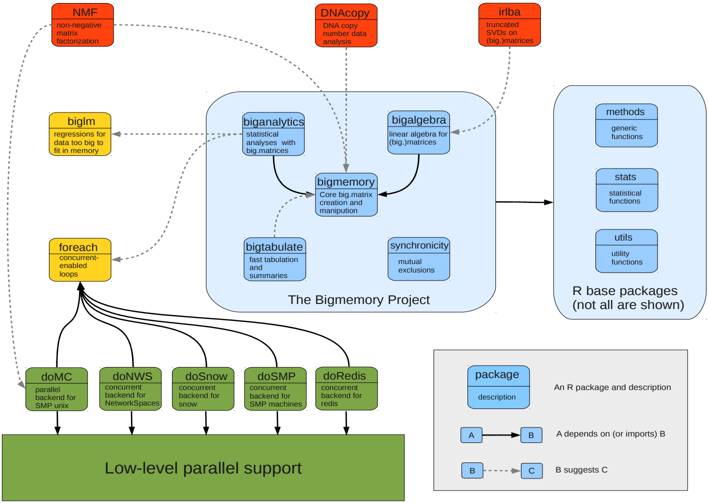
Why R?

R is the *lingua franca* of statistics:

- The syntax is simple and well-suited for data exploration and analysis.
- It has excellent graphical capabilities.
- It is extensible, with over 2500 packages available on CRAN alone.
- It is open source and freely available for Windows/MacOS/Linux platforms.

Currently, the Bigmemory Project is designed to extend the R programming environment through a set of packages (**bigmemory**, **bigtabulate**, **biganalytics**, **synchronicity**, and **bigalgebra**), but it could also be used as a standalone C++ library or with other languages and programming environments.

The Bigmemory Project: http://www.bigmemory.org/



In a nutshell...

- The approaches adopted by statisticians in analyzing small data sets don't scale to massive ones.
- Statisticians who want to explore massive data must
  - be aware of the various pitfalls;
  - adopt new approaches to avoid them.
- We will
  - illustrate common challenges for dealing with massive data;
  - provide general solutions for avoiding the pitfalls.

Importing and managing massive data

The Netflix data $\sim$ 2 GB: `read.table()` uses 3.7 GB total in the process of importing the data (taking 140 seconds):

```
> net <- read.table("netflix.txt", header=FALSE,
+                   sep="\t", colClasses = "integer")
> object.size(net)
1981443344 bytes
```

With bigmemory, only the 2 GB is needed (no memory overhead, taking 130 seconds):

```
> net <- read.big.matrix("netflix.txt", header=FALSE,
+                        sep="\t", type = "integer",
> net[1:3,]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    3 2005    9
[2,]    1    2    5 2005    5
[3,]    1    3    4 2005   10
```

Importing and managing massive data

- `read.table()`:
  - memory overhead as much as 100% of size of data
  - `data.frame` and `matrix` objects not available in shared memory
  - limited in size by available RAM, recommended maximum 10%-20% of RAM
- `read.big.matrix()`:
  - matrix-like data (not data frames)
  - no memory overhead
  - faster that `read.table()`
  - supports shared memory for efficient parallel programming
  - supports file-backed objects for data larger-than-RAM
- Databases and other alternatives:
  - Slower performance, no formal shared memory
  - Not compatible with linear algebra libraries
  - Require customized coding (chunking algorithms, generally)

Importing and managing massive data

With the full Airline data ($\sim$ 12 GB), **bigmemory**'s file-backing allows you to work with the data even with only 4 GB of RAM, for example:

```
> x <- read.big.matrix("airline.csv", header=TRUE,
+                      backingfile="airline.bin",
+                      descriptorfile="airline.desc",
+                      type="integer")
> x
An object of class "big.matrix"
Slot "address":
<pointer: 0x3031fc0>
> rm(x)
> x <- attach.big.matrix("airline.desc")
> dim(x)
[1] 123534969        29
```

Exploring massive data

```
> summary(x[, "DepDelay"])
       Min.      1st Qu.       Median         Mean
  -1410.000       -2.000        0.000        8.171
    3rd Qu.         Max.         NA's
      6.000     2601.000  2302136.000
>
> quantile(x[, "DepDelay"],
+          probs=c(0.5, 0.9, 0.99), na.rm=TRUE)
50% 90% 99%
  0  27 128
```

Example: caching in action

In a fresh R session on this laptop, newly rebooted:

```
> library(bigmemory)
> library(biganalytics)
> setwd("/home/jay/Desktop/BMPtalks")
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> system.time( numplanes <- colmax(x, "TailNum",
+                                  na.rm=TRUE) )
   user  system elapsed
  0.770   0.550   6.144
> system.time( numplanes <- colmax(x, "TailNum",
+                                  na.rm=TRUE) )
   user  system elapsed
  0.320   0.000   0.355
```

Split-apply-combine

- Many computational problems in statistics are solved by performing the same calculation repeatedly on independent sets of data. These problems can be solved by
  - partitioning the data (the *split*)
  - performing a single calculation on each partition (the *apply*)
  - returning the results in a specified format (the *combine*)
- Recent attention: it can be particularly efficient and easily lends itself to parallel computing
- "split-apply-combine" was coined by Hadley Wickham, but the approach has been supported on a number of different environments for some time under different names:
  - SAS: `by`
  - Google: `MapReduce`
  - Apache: `Hadoop`

Aside: split()

```
> x <- matrix(c(rnorm(5), sample(c(1, 2), 5,
+     replace = T)), 5, 2)
> x
           [,1] [,2]
[1,] -0.89691455    2
[2,]  0.18484918    1
[3,]  1.58784533    2
[4,] -1.13037567    1
[5,] -0.08025176    1
> split(x[, 1], x[, 2])
$`1`
[1]  0.18484918 -1.13037567 -0.08025176

$`2`
[1] -0.8969145  1.5878453
```

Exploring massive data

```
> GetDepQuantiles <- function(rows, data) {
+   return(quantile(data[rows, "DepDelay"],
+            probs=c(0.5, 0.9, 0.99), na.rm=TRUE))
+ }
>
> groups <- split(1:nrow(x), x[,"DayOfWeek"])
>
> qs <- sapply(groups, GetDepQuantiles, data=x)
>
> colnames(qs) <- c("Mon", "Tue", "Wed", "Thu",
+                   "Fri", "Sat", "Sun")
> qs
    Mon Tue Wed Thu Fri Sat Sun
50%   0   0   0   0   0   0   0
90%  25  23  25  30  33  23  27
99% 127 119 125 136 141 116 130
```

Aside: foreach

The user may register any one of several “parallel backends” like **doMC**, or none at all. The code will either run sequentially or will make use of the parallel backend, without modification.

```
> library(foreach)
>
> library(doMC)
> registerDoMC(2)
>
> ans <- foreach(i = 1:10, .combine = c) %dopar%
+     {
+         i^2
+     }
>
> ans
 [1]   1   4   9  16  25  36  49  64  81 100
```

Concurrent programming with **foreach** (1995 only)

- Such split-apply-combine problems can be done in parallel.
- Here, we start with one year of data only, 1995.
- Why only 1995? Memory implications.
- The message: shared memory is essential.

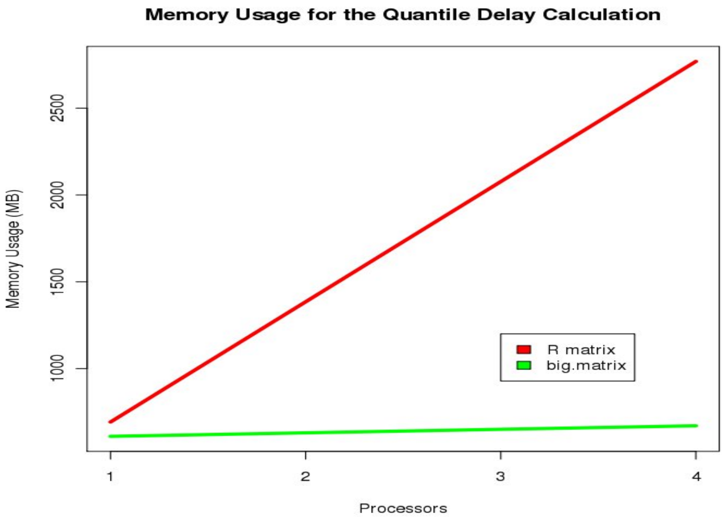
```
> library(foreach)
> library(doSNOW)
> cl <- makeSOCKcluster(4)
> registerDoSNOW(cl)
>
> x <- read.csv("1995.csv")
> dim(x)
[1] 5327435      29
```

Concurrent programming with **foreach** (1995 only)

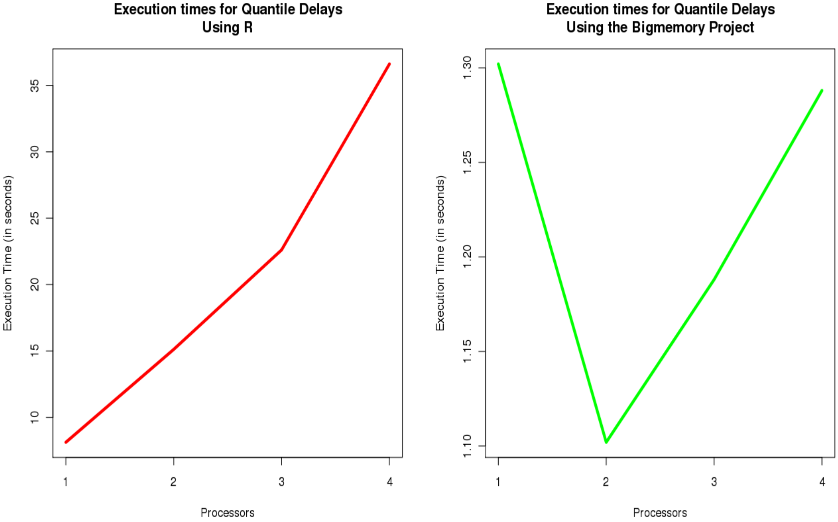
4 cores: 37 seconds; 3 cores: 23 seconds; 2 cores: 15 seconds, 1 core: 8 seconds. Memory overhead of 4 cores: > 2.5 GB!

```
> groups <- split(1:nrow(x), x[,"DayOfWeek"])
>
> qs <- foreach(g=groups, .combine=rbind) %dopar% {
+   GetDepQuantiles(g, data=x)
+ }
>
> daysOfWeek <- c("Mon", "Tues", "Wed", "Thu",
+                 "Fri", "Sat", "Sun")
> rownames(qs) <- daysOfWeek
> t(qs)
    Mon Tues Wed Thu Fri Sat Sun
50%   0    0   0   1   1   0   1
90%  21   21  26  27  29  23  23
99% 102  107 116 117 115 104 102
```

1995 delays in parallel: shared memory essential



1995 delays in parallel: shared memory essential



Aside: mwhich()

`mwhich()` works with either regular R matrices or with `big.matrix` objects (a neat trick for developers of new functions):

```
> x <- matrix(c(rnorm(5), sample(c(1, 2), 5,
+     replace = T)), 5, 2)
> x
           [,1] [,2]
[1,]  1.0517744    1
[2,] -0.7526655    1
[3,] -1.4396768    1
[4,] -0.2857115    2
[5,] -1.0342851    1
> mwhich(x, c(1, 2), list(0, 1),
+        list("le", "eq"), "AND")
[1] 2 3 5
```

Challenge: find plane “birthdays”

10 planes only (there are > 13,000 total planes).

With 1 core: 74 seconds. With 2 cores: 38 seconds.

All planes, 4 cores: $\sim$ 2 hours.

```
> library(foreach)
> library(doMC)
> registerDoMC(cores=2)
> planeStart <- foreach(i=1:10, .combine=c) %dopar% {
+
+   x <- attach.big.matrix(xdesc)
+   yearInds <- mwhich(x, "TailNum", i, comps="eq")
+   y <- x[yearInds,c("Year", "Month")]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   return(12*minYear + minMonth)
+
+ }
```

A better solution: split-apply-combine!

```
> birthmonth <- function(y) {
+   minYear <- min(y[,'Year'], na.rm=TRUE)
+   these <- which(y[,'Year']==minYear)
+   minMonth <- min(y[these,'Month'], na.rm=TRUE)
+   return(12*minYear + minMonth)
+ }
>
> time.0 <- system.time( {
+   planemap <- split(1:nrow(x), x[,"TailNum"])
+   planeStart <- sapply( planemap,
+     function(i) birthmonth(x[i, c('Year','Month'),
+                                 drop=FALSE]) )
+ } )
>
> time.0
   user  system elapsed
 53.520   2.020  78.925
```

Parallel split-apply-combine

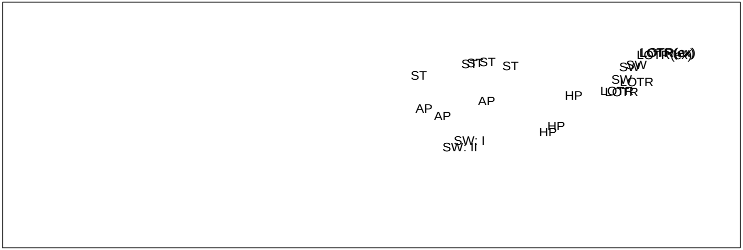
Using 4 cores, we can reduce the time to $\sim$ 20 seconds (not including the `read.big.matrix()`, repeated here for a special reason:

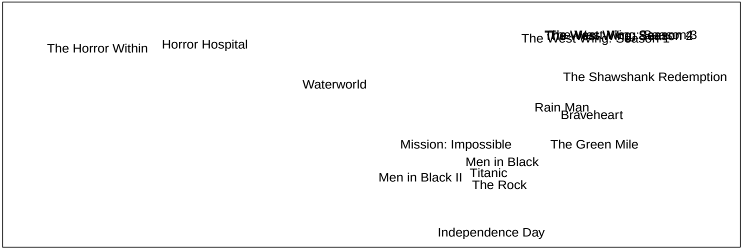
```
x <- read.big.matrix("airline.csv", header = TRUE,
                     backingfile = "airline.bin",
                     descriptorfile = "airline.desc",
                     type = "integer",
                     extraCols = "Age")
planeindices <- split(1:nrow(x), x[, "TailNum"])
planeStart <- foreach(i = planeindices,
                      .combine = c) %dopar% {
  birthmonth(x[i, c("Year", "Month"), drop = FALSE])
}
x[, "Age"] <- x[, "Year"] * as.integer(12) +
              x[, "Month"] -
              as.integer(planeStart[x[, "TailNum"]])
```

Massive linear models via **biglm**

```
> library (biganalytics)
> x <- attach.big.matrix("airline.desc")
> blm <- biglm.big.matrix(DepDelay ~ Age, data = x)
> summary(blm)
Large data regression model: biglm(formula = formula,
  data = data , ...)
Sample size = 84406323
              Coef   (95%    CI)      SE p
(Intercept) 8.5889 8.5786 8.5991 0.0051 0
Age         0.0053 0.0051 0.0055 0.0001 0
```

Netflix: a truncated singular value decomposition





Concluding example: satellite images

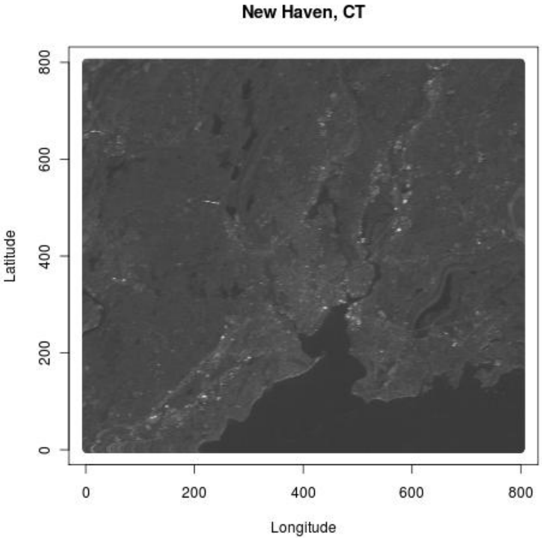
```
read.landsat <- function(file, descfile, dims=NULL,
                         type="char") {
  if (is.null(dims)) stop("dimensions must be known")
  if (length(grep(file, dir()))==0)
    stop("file does not exist")
  x <- big.matrix(1, 1, type=type, backingfile="X.bin",
                  descriptorfile="X.desc")
  xdesc <- dget("X.desc")
  xdesc@description$filename <- file
  xdesc@description$totalRows <- prod(dims)
  xdesc@description$nrow <- prod(dims)
  dput(xdesc, descfile)
  x <- as(attach.big.matrix(descfile), "big.3d.array")
  x@dims <- dims
  return(x)
}
```

Concluding example: satellite images

```
x <- read.landsat(file = "Landsat_17Apr2005.dat",
                  descfile = "Landsat_17Apr2005.desc",
                  dims = c(800,800,6))

plot(rep(1:800, 800), rep(800:1, each=800),
     col=gray(x[,,1]/256),
     xlab="Longitude", ylab="Latitude",
     main="New Haven, CT")
```

Sample image (apologies for the resolution)



Summary

The Bigmemory Project proposes three new ways to work with very large sets of data:

- memory and file-mapped data structures, which provide access to arbitrarily large sets of data while retaining a look and feel that is familiar to statisticians;
- data structures that are shared across processor cores on a single computer, in order to support efficient parallel computing techniques when multiple processors are used;
- and file-mapped data structures that allow concurrent access by the different nodes in a cluster of computers.

Even though these three techniques are currently implemented only for R, they are intended to provide a flexible framework for future developments in the field of statistical computing.

Thanks! http://www.bigmemory.org/

- Dirk Eddelbuettel, Bryan Lewis, Steve Weston, and Martin Schultz, for their feedback and advice over the last three years
- Bell Laboratories (Rick Becker, John Chambers and Allan Wilks), for development of the S language
- Ross Ihaka and Robert Gentleman, for their work and unselfish vision for R
- The R Core team
- David Pollard, for pushing us to better communicate the contributions of the project to statisticians
- John Hartigan, for years of teaching and mentoring
- John Emerson (my father, Middlebury College), for getting me started in statistics
- Many of my students, for their willingness argue with me

OTHER SLIDES

`http://www.bigmemory.org/`

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

and

`...yale.edu/~jay/Brazil/SP/bigmemory/`

Overview: the Bigmemory Project

- Problems and challenges:
  - R frequently makes copies of objects, which can be costly
  - Guideline: R's performance begins to degrade with objects more than about 10% of the address space, or when total objects consume more than about 1/3 of RAM.
  - swapping: not sufficient
  - chunking: inefficient, customized coding
  - parallel programming: memory problems, non-portable solutions
  - shared memory: essentially inaccessible to non-experts

- Key parts of the solutions:
  - operating system caching
  - shared memory
  - file-backing for larger-than-RAM data
  - a framework for platform-independent parallel programming (credit Steve Weston, independently of the BMP)

Extending capabilities versus extending the language

- Extending R's capabilities:
  - Providing a new algorithm, statistical analysis, or data structure
  - Examples: `lm()`, or package **bcp**
  - Most of the 2500+ packages on CRAN and elsewhere

- Extending the language:
  - Example: **grDevices**, a low-level interface to create and manipulate graphics devices
  - Example: **grid** graphics
  - The packages of the Bigmemory Project:
    - a developer's interface to underlying operating system functionality which could not have been written in R itself
    - a higher-level interface designed to mimic R's matrix objects so that statisticians can use their current knowledge of computing with data sets as a starting point for dealing with massive ones.