Massive data, shared and distributed memory, and concurrent programming: bigmemory and foreach

Michael J Kane
John W. Emerson

Department of Statistics
Yale University

- Flight arrival and departure details for all* commercial flights within the USA, from October 1987 to April 2008.
- Nearly 120 million records, 29 variables (mostly integer-valued)
- We preprocessed the data, creating a single CSV file, recoding the carrier code, plane tail number, and airport codes as integers.

* Not really. Only for carriers with at least 1% of domestic flights in a given year.
Hardware used in the examples

Yale’s “Bulldogi” cluster:
- 170 Dell Poweredge 1955 nodes
- 2 dual-core 3.0 Ghz 64-bit EM64T CPUs
- 16 GB RAM each node
- Gigabit ethernet with both NFS and a Lustre filesystem
- Managed via PBS

A nice laptop (it ain’t light):
- Dell Precision M6400
- Intel Core 2 Duo Extreme Edition
- 4 GB RAM (a deliberate choice)
- Plain-vanilla primary hard drive
- 64 GB solid state secondary drive
120 million flights by 29 variables ~ 3.5 billion elements. Too big for an R matrix (limited to $2^{31} - 1 \sim 2.1$ billion elements and likely to exceed available RAM, anyway).

Hadley Wickham’s recommended approach: *sqlite*

An alternative to *bigmemory*: *ff*

- We used version 2.1.0 (beta)
- *ff* matrix limited to $2^{31}-1$ elements;
- *ffdf* data frame works, though.

Others: *BufferedMatrix, filehash*,

- many database interface packages;
- *R.huge* will no longer be supported.
Via `bigmemory` (on CRAN): creating the filebacked `big.matrix`

Note: as part of the creation, I add an extra column that will be used for the calculated age of the aircraft at the time of the flight.

```r
> x <- read.big.matrix("AirlineDataAllFormatted.csv", header=TRUE, type="integer", backingfile="airline.bin", descriptorfile="airline.desc", extraCols="age")
```

~ 25 minutes
Via SQLite (http://sqlite.org/): preparing the database

```
Revo$ sqlite3 ontime.sqlite3
SQLite Version 3.6.10 ...
sqlite> create table ontime (Year int, Month int, ...
    _, origin int, _, LateAircraftDelay int);
sqlite> .separator ,
sqlite> .import AirlineDataAllFormatted.csv ontime
sqlite> delete from ontime where typeof(year)=="text";
sqlite> create index origin on ontime(origin);
sqlite> .quit
Revo$
```

~ 75 minutes excluding the create index.
A first comparison: bigmemory vs RSQLite

Via RSQLite and bigmemory, a column minimum? The result: bigmemory wins.

```r
> library(bigmemory)
> x <- attach.big.matrix(dget("airline.desc") )
> system.time(colmin(x, 1))
user  system elapsed
0.236   0.372   7.564
> system.time(a <- x[,1])
user  system elapsed
0.852   1.060   1.910
> system.time(a <- x[,2])
user  system elapsed
0.800   1.508   9.246
> library(RSQLite)
> x <- attach.big.matrix(dget("airline.desc") )
> ontime <- dbConnect("SQLite",
+       dbname="ontime.sqlite3")
> from_db <- function(sql) {
+       dbGetQuery(ontime, sql)
+   }
> system.time(from_db(  
+       "select min(year) from ontime")
+   )
user  system elapsed
45.722  14.672  129.098
> system.time(a <-
+    from_db("select year from ontime"))
user  system elapsed
59.208  20.322  138.132
```
Example: *ff* (Dan Adler et.al., Beta version 2.1.0)

```r
> library(bigmemory)
> library(filehash)
> x <- attach.big.matrix(dget("airline.desc"))
> y1 <- ff(x[,1], filename="ff1")
> y2 <- ff(x[,2], filename="ff2")
...
> y30 <- ff(x[,30], filename="ff30")
> z <- ffdf(y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,
+ y11,y12,y13,y14,y15,y16,y17,y18,y19,y20,
+ y21,y22,y23,y24,y25,y26,y27,y28,y29,y30)
```

With apologies to Adler et.al, we couldn’t figure out how to do this more elegantly, but it worked (and, more quickly – 7 minutes, above – than you’ll see with the subsequent two examples with other packages). As we noted last year at UseR!, a function like `read.big.matrix()` would greatly benefit *ff*. 
Airline on-time performance via \textit{ff}

Example: \textit{ff} (Dan Adler et.al., Beta version 2.1.0)

The challenge: R’s \textit{min()} on extracted first column; caching.

The result: they’re about the same.

\begin{verbatim}
# With ff:
> system.time(min(z[,1], na.rm=TRUE))
  user  system elapsed
 2.188   1.360   10.697
> system.time(min(z[,1], na.rm=TRUE))
  user  system elapsed
 1.504   0.820   2.323

> # With bigmemory:
> system.time(min(x[,1], na.rm=TRUE))
  user  system elapsed
 1.224   1.556   10.101
> system.time(min(x[,1], na.rm=TRUE))
  user  system elapsed
 1.016   0.988   2.001
\end{verbatim}
Airline on-time performance via \textit{ff}

Example: \textit{ff} (Dan Adler et.al., Beta version 2.1.0)

The challenge: alternating \texttt{min()} on first and last rows.

The result: maybe an edge to \textit{bigmemory}, but do we care?

\begin{tabular}{llll}
  \texttt{min(z[1,], na.rm=TRUE)} & \texttt{min(z[nrow(z),], na.rm=TRUE)} & \texttt{min(z[1,], na.rm=TRUE)} & \texttt{min(z[nrow(z),], na.rm=TRUE)} \\
  0.040 & 0.000 & 0.115 & 0.032 & 0.000 & 0.099 & 0.020 & 0.000 & 0.024 & 0.036 & 0.000 & 0.080
\end{tabular}
Airline on-time performance via `ff`

Example: `ff` (Dan Adler et.al., Beta version 2.1.0)

The challenge: random extractions, two runs (time two):

```r
> theserows <- sample(nrow(x), 10000)
> thesecols <- sample(ncol(x), 10)

> # With ff:
> system.time(a <- z[theserows, +
                   thesecols])

user  system elapsed
0.092   1.796  60.574

> system.time(a <- z[theserows, +
                   thesecols])

user  system elapsed
0.040   0.384  4.069

> # With bigmemory:
> system.time(a <- x[theserows, +
                   thesecols])

user  system elapsed
0.020   1.612  64.136

> system.time(a <- x[theserows, +
                   thesecols])

user  system elapsed
0.020   0.024  1.323

> theserows <- sample(nrow(x), 100000)
> thesecols <- sample(ncol(x), 10)

> # With ff:
> system.time(a <- z[theserows, +
                   thesecols])

user  system elapsed
0.352   3.305  78.161

> system.time(a <- z[theserows, +
                   thesecols])

user  system elapsed
0.340   3.156  77.623

> # With bigmemory:
> system.time(a <- x[theserows, +
                   thesecols])

user  system elapsed
0.248   2.752  78.935

> system.time(a <- x[theserows, +
                   thesecols])

user  system elapsed
0.248   2.676  78.973
```

Laptop
Airline on-time performance via *filehash*

Example: *filehash* (Roger Peng, on CRAN)

```r
> library(bigmemory)
> library(filehash)
> x <- attach.big.matrix(dget("airline.desc"))
> dbCreate("filehashairline", type="RDS")
> fhdb <- dbInit("filehashairline", type="RDS")
> for (i in 1:ncol(x))
+   dbInsert(fhdb, colnames(x)[i], x[,i])   # About 15 minutes.
```

```r
> system.time(min(x[,"Year"]))
user  system elapsed
11.817   0.236  11.584
> system.time(colmin(x, "Year"))
user  system elapsed
0.184   0.000   0.183
```

*filehash* is quite memory-efficient on disk!
Airline on-time performance via *BufferedMatrix*

**Example: BufferedMatrix** (Ben Bolstad, on BioConductor)

```r
> library(bigmemory)
> library(BufferedMatrix)
> x <- attach.big.matrix(dget(“airline.desc”))
> y <- createBufferedMatrix(nrow(x), ncol(x))
> for (i in 1:ncol(x)) y[,i] <- x[,i]
```

More than 90 minutes to fill the *BufferedMatrix*; inefficient (only 8-byte numeric is supported); not persistent.

```r
> system.time(colmin(x))
user  system elapsed
4.576   4.560  113.289
> system.time(colMin(y))
user  system elapsed
20.926  71.492  966.952
```

```r
> system.time(colmin(x, na.rm=TRUE))
user  system elapsed
11.264   9.645  256.911
> system.time(colMin(y, na.rm=TRUE))
user  system elapsed
39.515  70.436  941.229
```
More basics of `bigmemory`

```r
> library(bigmemory)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> dim(x)
[1] 118914548       30
> colnames(x)
[1] "Year" "Month" "DayofMonth" "DayOfWeek" "DepTime" "CRSDepTime"
[7] "ArrTime" "CRSArrTime" "UniqueCarrier" "FlightNum" "TailNum" "ActualElapsedTime"
[10] "ArrDelay" "Origin" "Dest"
... (rows omitted for this slide)
> tail(x, 1)
          Year Month DayofMonth DayOfWeek
2008       4      17           4
DepTime    381  CRSDepTime  ArrTime  CRSArrTime
          375      472         754
UniqueCarrier  11  FlightNum  TailNum ActualElapsedTime
            1211      2057           91
CRSElapsedTime  99  AirTime  ArrDelay DepDelay
             64       -2           6
Origin      63        Dest  Distance TaxiIn
           35        430        15
... (rows omitted for this slide)
```
More basics of `bigmemory`

```r
> # Can we get all flights from JFK to SFO?  Sure!
>
> a <- read.csv("AirportCodes.csv")
> a <- na.omit(a)
> JFK <- a$index[a$airport=="JFK"]
> SFO <- a$index[a$airport=="SFO"]
>
> gc(reset=TRUE)
> system.time(
+   y <- x[x[,"Origin"]==JFK & x[,"Dest"]==SFO,]
+ )
> dim(y)
[1] 99867  30
> gc()
> rm(y)
```

Slower and less memory-efficient than our alternative: `mwhich()`, coming up next...
mwhich()

```r
> # mwhich() for fast, no-overhead "multi-which"
>
> gc(reset=TRUE)

used (Mb) gc trigger (Mb) max used (Mb)
Ncells 214238 11.5 407500  21.8  214238 11.5
Vcells 169034  1.3 176605889 1347.4  169034  1.3
>
> system.time(
+   y <- x[mwhich(x, cols=c("Origin", "Dest"),
+     vals=list(JFK, SFO),
+     comps="eq",
+     op="AND"), ]
+ )

user  system elapsed
 5.270   0.020   5.308
>
> dim(y)
[1] 99867    30

Fast, no memory overhead!
```

Laptop
mwhich(): useful with R matrices, too!

```r
> # mwhich() works on a matrix, too, but I can't
> # hold all the data as an R matrix, even if I had
> # the RAM (see earlier comment on size). On a subset:
>
> xx <- x[,15:18]
> gc(reset=TRUE)

used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells   203561   10.9     407500   21.8    203561   10.9
Vcells 237996106 1815.8  499861463 3813.7 237996106 1815.8

> system.time(
+   y <- xx[mwhich(xx, cols=c("Origin", "Dest"),
+ vals=list(JFK, SFO),
+ comps="eq",
+ op="AND"), ]
+ )

user  system elapsed
5.220   0.000   5.219

> dim(y)
[1] 99867     4
> gc()

used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells   203566   10.9     407500   21.8    213419   11.4
Vcells 238195846 1817.3  499861463 3813.7 238448239 1819.3
```

Just as fast as with a `big.matrix`, with no memory overhead beyond the matrix itself.
For each plane in the data set, what was the first month (in months A.D.) of service?

No surprises... yet.
Introducing *foreach, iterators, doMC, doSNOW, doNWS*

- The brainchildren of Steve Weston

- The following are called “parallel backends”:
  - *doMC* makes use of *multicore* (Simon Urbanek)
  - *doNWS* makes use of NetWorkSpaces (*nws*, REvolution Computing following from Scientific Computing Associates)
The foreach package

• Provides a new looping construct that supports parallel computing
• Hybrid between for-loop and lapply
• Depends on the iterators package
• Similar to foreach and list comprehensions in Python and other languages
Why should you care?

- Not just another parallel programming package
- It’s a framework that allows R code to be easily executed in parallel with different parallel backends
- Currently support with NWS, Snow, multicore
- Adding support for Hadoop, Windows shared memory
It makes easy things simple

- Works like `lapply()`, but you don’t have to specify the processing as a function
- Use `%do%` to execute sequentially
- Use `%dopar%` to execute in parallel (possibly)

```r
x <- foreach(i=1:10) %dopar% {
  sqrt(i)
}
```
Data automatically exported

• Body of foreach is analyzed and variables needed from the local environment are automatically exported

```r
m <- matrix(rnorm(16), 4, 4)
foreach(i=1:ncol(m)) %dopar% {
  mean(m[,i])
}
```
foreach on SMP via doMC (master plus 4 workers)

### Package `multicore` by Simon Urbanek

- Very new!
- ~ 2.5 hours
- A new type of loop structure
- Some initialization

```r
> date()
> library(bigmemory)
> library(doMC)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1

> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+ require(bigmemory)
+ x <- attach.big.matrix(xdesc)
+ y <- x[mwhich(x, "TailNum", i, 'eq'),
+        c("Year", "Month"), drop=FALSE]
+ minYear <- min(y,"Year"), na.rm=TRUE)
+ these <- which(y,"Year"==minYear)
+ minMonth <- min(y[these,"Month"], na.rm=TRUE)
+ 12*minYear + minMonth
+ }
> date()
[1] "Fri Jun 19 00:14:36 2009"
```
foreach on SMP via doSNOW (master plus 3 workers)

> date()
[1] "Fri Jun 19 09:10:22 2009"
> library(bigmemory)
> library(doSNOW)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: snow
> cl <- makeSOCKcluster(3)
> registerDoSNOW(cl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+         c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }

> date()

• ~ 3.5 hours

• Different parallel backend setup and registration

• Otherwise identical code to the doMC SMP version

foreach on SMP via doNWS (master plus 3 workers)

> date()
> library(bigmemory)
> library(doNWS)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: nws
> sl <- sleigh(workerCount=3)
> registerDoNWS(sl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+   c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y,"Year"==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> date()

- ~ 3.5 hours
- A different parallel backend setup and registration
- Otherwise identical code to the doMC and doSNOW SMP versions
`foreach` on cluster via `doNWS` (10 nodes by 3 processors)

```r
> date()
> library(bigmemory)
> library(doNWS)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: nws
> nodes <- pbsNodeList()[-1]
> sl <- sleigh(nodeList=nodes, launch=sshcmd)
> registerDoNWS(sl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[, "TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+         c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[, "Year"], na.rm=TRUE)
+   these <- which(y[, "Year"]==minYear)
+   minMonth <- min(y[these, "Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> dput(planeStart, "planeStart30NWS.txt")
> date()
```

# Cluster Setup:

- qsub -I -l nodes=10:ppn=3 -q sandbox
- Once launched, fire up R on master.

• ~ 40 minutes (slower than expected – why?)

• No substantive code changes from the SMP version

• Different `sleight()` (NetWorkSpaces) setup for cluster
Big \texttt{big.matrix}: no $2^{31}$ row limitation

\begin{verbatim}
> R <- 3e9          # 3 billion rows
> C <- 2            # 2 columns
>
> R*C*8             # 48 GB total size
[1] 4.8e+10
>
> date()

> x <- filebacked.big.matrix(R, C, type='double',
+                            backingfile='test.bin',
+                            descriptorfile='test.desc')

> x[1,] <- rnorm(C)
> x[nrow(x),] <- runif(C)
> summary(x[1,])
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.7510 -1.2640 -0.7777 -0.7777 -0.2912  0.1953
> summary(x[nrow(x),])
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.04232 0.21080 0.37930 0.37930 0.54780 0.71630

> date()
\end{verbatim}
The new package *synchronicity*

- Locking has been removed from *bigmemory* itself (upcoming version 4.0 and onwards) so that packages can take advantage of synchronization mechanisms without having to install bigmemory.
  - exclusive locks
  - shared locks
  - timed locks
  - conditional locking
- Allows for the creation of Universal Unique Identifiers
- The following locking schemes have been implemented for use in *bigmemory* (version 4.0 and onwards).
  - no locking
  - read only (allows a *big.matrix* object to be read only)
  - column locking
  - row locking
- The architecture is flexible enough to allow a user to define his own mutex scheme for a *big.matrix* object.
Supporting linear algebra routines with *bigalgebra*

- **bigalgebra** (currently in development) will support linear algebra operations on *R* matrices as well as *big.matrix* objects (including various combinations) for the following operations:
  
  - matrix copy
  - scalar multiply
  - matrix addition
  - matrix multiplication
  - SVD
  - eigenvalues and eigenvectors
  - Cholesky factorization
  - QR factorization
  - others?

- The routines are implemented in BLAS and LAPACK libraries
In summary: *bigmemory* and more

- User-friendly, familiar interface (less user overhead than the other alternatives)
- Memory-efficient externalities (e.g. `mwhich()` cleverness)
- Shared memory will full mutexes (SMP)
- Distributed memory on filesystems supporting mmap
- A developer tool, with access to pointers in C++ allowing integration with existing libraries (e.g. linear algebra routines).

- *foreach* plus *bigmemory*: a winning combination for massive data concurrent programming