The bigmemoRy package: handling large data sets in R using RAM and shared memory

John W. Emerson[1], **Michael J. Kane**

Yale University

Abstract

Multi-gigabyte data sets challenge and frustrate **R** users even on well-equipped hardware. **C** programming can be helpful, but is cumbersome for interactive data analysis and lacks the flexibility and power of **R**'s rich statistical programming environment. The new package *bigmemoRy* bridges this gap, implementing massive matrices in memory (managed in **R** but implemented in **C**) and supporting their basic manipulation and exploration. It is ideal for problems involving the analysis in **R** of manageable subsets of the data, or when an analysis is conducted mostly in **C**. In a Unix environment, the data structure may be allocated to shared memory, allowing separate **R** processes on the same computer to share access to a single copy of the data set. This opens the door for more powerful parallel analyses and data mining of massive data sets.

[1]Corresponding author: John W. Emerson, email: john.emerson@yale.edu

Contents

1 Introduction

A numeric matrix containing 100 million rows and 5 columns consumes approximately 4 gigabytes (GB) of memory in the **R** statistical programming environment (9). Such massive, multi-gigabyte data sets challenge and frustrate **R** users even on well-equipped hardware. Even moderately large data sets can be problematic; guidelines on **R**'s native capabilities are discussed in the installation manual (10). The **C** programming language allows quick, memory-efficient operations on massive data sets, without the memory overhead of many **R** operations. Unfortunately, the **C** language is not well-suited for interactive data exploration, lacking the flexibility, power, and convenience of **R**'s rich environment.

The new package *bigmemoRy* bridges the gap between **R** and **C**, implementing massive matrices in memory and supporting their basic manipulation and exploration. Version 1.0 supports matrices of double, integer, short, and char data types. In *nix environments, the package supports the use of shared memory for matrices. An API is also provided, allowing savvy **C** programmers to extend the functionality of *bigmemoRy*.

As of 2008, typical high-end personal computers (PCs) have 1-4 GB of random access memory (RAM) and run 32-bit operating systems. A small number of PCs might have more than 4 GB of memory and 64-bit operating systems, and such configurations are now common on workstations, servers and high-performance computing clusters. At Google, for example, Daryl Pregibon's group uses 64-bit Linux workstations with up to 32 GB of RAM. His group studies massive subsets of terabytes (though perhaps not googols) of data. Massive data sets are increasingly common; the Netflix Prize[2] competition involves the analysis of approximately 100 million movie ratings, and the basic data structure would be a 100 million by 5 matrix of integers (movie ID, customer ID, rating, rental year and month).

Data frames and matrices in **R** were designed for data sets much smaller in size than the computer's memory limit. They are flexible and easy to use, with typical manipulations executing quickly on smaller data sets. They suit the needs of the vast majority of **R** users and work seamlessly with existing **R** functions and packages. Problems arise, however, with larger data sets; we provide a brief discussion in the appendix.

A second category of data sets are those requiring more memory than a machine's RAM. CRAN and Bioconductor packages such as *DBI*, *RJDBC*, *RMySQL*, *RODBC*, *ROracle*, *TSMySQL*, *filehashSQLite*, *TSSQLite*, *pgUtils*, and *Rdbi* allow users to extract subsets of traditional databases using SQL statements. Other packages, such as *filehash* and *ff*, provide a convenient `data.frame`-like interface. As noted by Adler *et. al.*, authors of the *ff* package (1), "the idea is that one can read from and write to" flat files or databases, "and operate on

[2] http://www.netflixprize.com

the parts that have been loaded into **R**." While they all help manage massive data sets, the user is often forced to wait for disk accesses, and none of these are well-suited to handling the synchronization challenges posed by concurrent programming.

We designed *bigmemoRy* to address a third category of data sets. These can be massive data sets (perhaps requiring several GB of memory on typical computers, as of 2008) but not larger than the total available RAM. In this case, disk accesses are unnecessary.[3] In some cases, a traditional data frame or matrix might suffice to store the data, but there may not be enough RAM to handle the overhead of working with a data frame or matrix. The appendix outlines some of **R**'s limitations for this type of data set. The `BigMatrix` class has been created to fill this niche, creating efficiencies with respect to data types and opportunities for parallel computing and analyses of massive data sets in RAM using **R**.

Fast-forward to year 2016, eight years hence. A naive application of Moore's Law projects a sixteen-fold increase (four doublings) in hardware capacity, although experts caution that "the free lunch is over" (12). They predict that further boosts in CPU performance will be limited, and note that manufacturers are turning to hyper-threading and multicore architectures, reviving interest in parallel computing. We designed *bigmemoRy* for the purpose of fully exploiting available RAM for large data analysis problems, and to facilitate concurrent programming using **R** and **C**.

With *bigmemoRy*, we can exploit simultaneously the strengths of **R** and **C**. Multiple processors on the same machine can share access to the same copy of the massive data set, and subsets of rows and columns may be extracted quickly and easily for standard analyses in **R**. Most significantly, **R** users of *bigmemoRy* don't need to be **C** experts (and don't have to use **C** at all, in most cases). And **C** programmers can make use of **R** as a convenient interface, without needing to become experts in the environment. Thus, *bigmemoRy* offers something for the demanding users and developers, extending and augmenting the **R** statistical programming environment for users with massive data sets and developers interested in concurrent programming with shared memory.

2 Using the bigmemoRy package

We use the Netflix Prize data as an example. The training set includes 99,072,112 ratings and five integer variables: movie ID, customer ID, rating, rental year and month. As a regular **R** numeric matrix, this would require approximately 4 GB of RAM, whereas only 2 GB is needed for the `BigMatrix` of integers. An integer matrix in **R** would be equally efficient, but

[3]We note that swap space could be used for matrices exceeding the available RAM. The performance will be degraded, however, and users may prefer to use one of available database solutions.

working with such a massive matrix in **R** would risk creating substantial memory overhead (see the appendix for a more complete discussion of the risks).

Our first example demonstrates only one new function, `read.bm()`; most **R** users are familiar with the three subsequent commands, `dim()`, `head()`, and `summary()`, implemented with new methods. We place the object in shared memory for convenience in subsequent examples.

```
> library(bigmemoRy)
> x <- read.bm("ALLtraining.txt", sep = "\t", type = "integer",
+     shared = TRUE, col.names = c("movie", "customer", "rating",
+         "year", "month"))
> dim(x)

[1] 99072112        5

> head(x)

     movie customer rating year month
[1,]     1        1      3 2005     9
[2,]     1        2      5 2005     5
[3,]     1        3      4 2005    10
[4,]     1        5      3 2004     5
[5,]     1        6      3 2005    11
[6,]     1        7      4 2004     8

> summary(x)

          min    max         mean NAs
movie       1  17770 9.100050e+03   0
customer    1 480189 1.297173e+05   0
rating      1      5 3.603304e+00   0
year     1999   2005 2.004245e+03   0
month       1     12 6.692275e+00   0
```

There are, in fact, 17770 movies in the Netflix data and 480,189 customers. Ratings range from 1 to 5 for rentals in 1999 through 2005. Standard **R** matrix notation is supported through the bracket operator.

```
> x[480185:480189, c("movie", "customer", "rating")]

     movie customer rating
[1,]   143    37314      5
[2,]   143    75110      3
[3,]   143    75075      4
[4,]   143   165558      4
[5,]   143   214342      5
```

One of the most important new functions is `which.bm()`. Based loosely on **R**'s `which()`, it provides high-performance comparisons with no memory overhead. Suppose we are interested in the ratings provided by customer number 50. For the `BigMatrix` created above, the logical expression `x[,2]==50` would extract the second column of the matrix as a massive numeric vector in **R**, do the logical comparison in **R**, and produce a massive **R** boolean vector. The command `which.bm(x, 2, 50, 'eq')` (or equivalently, `which.bm(x, 'customer', 50, 'eq')`) produces no memory overhead in **C** and returns only a vector of indices of length `sum(x[,2]==50)`.

```
> cust.indices.inefficient <- which(x[, "customer"] == 50)
> cust.indices <- which.bm(x, "customer", 50, "eq")
> length(cust.indices.inefficient)

[1] 616

> length(cust.indices)

[1] 616

> head(x[cust.indices, ])

     movie customer rating year month
[1,]     1       50      3 2004     5
[2,]    30       50      3 2004     5
[3,]    58       50      4 2004     9
[4,]    68       50      4 2004    12
[5,]    84       50      4 2004     8
[6,]   169       50      3 2004     8
```

More complex comparisons are supported by `which.bm()`, including the specification of minimum and maximum test values and comparisons on multiple columns in conjunction with `AND` and `OR` operations. For example, we might be interested in customer 50's movies which were rated 2 or worse during February through October of 2004:

```
> these <- which.bm(x, c(2, 4, 5, 3), list(50, 2004, c(2, 10),
+     2), list("eq", "eq", c("ge", "le"), "le"), "AND")
> x[these, ]
```

```
     movie customer rating year month
[1,]  1560       50      2 2004    10
[2,]  1865       50      2 2004     9
[3,]  4525       50      1 2004     3
[4,] 10583       50      2 2004     5
[5,] 10867       50      1 2004     9
[6,] 13558       50      2 2004     2
```

```
> mnames <- read.csv("movie_titles.txt", header = FALSE)
> names(mnames) <- c("movie", "year", "Name of Movie")
> mnames[mnames[, 1] %in% unique(x[these, 1]), c(1, 3)]
```

```
      movie                                          Name of Movie
1587   1560 Disney Princess Stories: Vol. 1: A Gift From the Heart
1899   1865                  Eternal Sunshine of the Spotless Mind
4611   4525                               Nick Jr. Celebrates Spring
10770 10583                                     The School of Rock
11061 10867    Disney Princess Party: Vol. 1: Birthday Celebration
13810 13558     An American Tail: The Mystery of the Night Monster
```

One of the authors thinks "The School of Rock" deserved better than a wimpy rating of 2; we haven't seen any of the others. Even more complex comparisons could involve set operations in **R** involving collections of indices returned by `which.bm()` from **C**.

The core functions supporting `BigMatrix` objects are:

`BigMatrix()`	`is.BigMatrix()`	`as.BigMatrix()`	`nrow()`	`deepcopy.bm()`
`ncol()`	`dim()`	`head()`	`tail()`	`colnames()`
`print()`	`which.bm()`	`read.bm()`	`write.bm()`	`rownames()`
`rm.cols.bm()`	`add.cols.bm()`	`hash.mat.bm()`	`dimnames()`	`"["` and `"[<-]"`

Other basic functions are included, useful by themselves and also serving as templates for the development of new functions. These include:

```
colmin()     min()        max()      colmax()
colrange()   range()      colmean()  mean()
colvar()     colsd()      summary()  biglm.bm()
bigglm.bm()  kmeans.bm()
```

2.1 Using Lumley's biglm package with bigmemoRy

Support for Thomas Lumley's *biglm* package (5) is provided via the `biglm.bm()` and `bigglm.bm()` functions; "biglm" stands for "bounded memory linear regression." In this example, the movie release year is used (as a factor) to try to predict customer ratings:

```
> lm.0 = biglm.bm(rating ~ year, data = x, fc = "year")
> print(summary(lm.0)$mat)
```

```
                  Coef        (95%         CI)           SE            p
(Intercept)  3.67616085  3.67586120  3.67646050 0.0001498258 0.000000e+00
year2004    -0.08152799 -0.08202262 -0.08103335 0.0002473167 0.000000e+00
year2003    -0.26993103 -0.27067766 -0.26918440 0.0003733163 0.000000e+00
year2002    -0.29444706 -0.29552627 -0.29336784 0.0005396078 0.000000e+00
year2001    -0.28545089 -0.28710257 -0.28379921 0.0008258412 0.000000e+00
year2000    -0.31096880 -0.31323569 -0.30870192 0.0011334430 0.000000e+00
year1999    -0.33915442 -0.38543277 -0.29287607 0.0231391736 1.212505e-48
```

It would appear that movie ratings provided in 2004 and 2005 movies were rated higher (on average) than rentals in earlier years. This particular regression will not win the $1,000,000 Netflix prize. However, it does illustrate the use of a `BigMatrix` to manage and study several gigabytes of data.

2.2 Shared memory

Here, we show how NetWorkSpaces (NWS, package *nws* (8)) and SNOW (package *snow*, for "small network of workstations" (11)) can be used for parallel computing using a shared `BigMatrix`. As noted earlier, future performance gains in statistical computing may depend more on software design and algorithms than on further advances in hardware. Adler *et. al.*

(1) encouraged **R** programmers to watch for opportunities for chunk-based processing, and opportunities for concurrent programming deserve similar attention.

First, we prepare a description of the shared object, containing necessary information about the matrix and the mutual exclusions (mutexes) that prevent read/write conflicts. The contents of the description are presented and discussed in section 3.

```
> xdescr <- DescribeBigSharedMatrix(x)
```

Next, we specify a "worker" function. In this simple example, its job is to attach the shared matrix and return the range of values in the specified column.

```
> worker <- function(i, descr.bm) {
+     require(bigmemoRy)
+     big <- AttachBigSharedMatrix(descr.bm)
+     return(colrange(big, cols = i))
+ }
```

Both the description (`xdesrc`) and the worker function (`worker()`) are used by ***nws*** and ***snow***, below. We conclude the section by illustrating a low-tech interactive use of shared memory, where the matrix description is passed between R sessions using a file.

2.2.1 Shared memory via Networkspaces

The following `sleigh()` command prepares the two "workers" on the local workstation, while `nwsHost` identifies the server which manages the Networkspaces communications (and this may or may not be the localhost, depending on the configuration). The result is a list with five ranges, one for each column of the matrix, and the results match those produced by `summary()` on page 4.

```
> library(nws)
> s <- sleigh(nwsHost = "HOSTNAME.stat.yale.edu", workerCount = 2)
> eachElem(s, worker, elementArgs = 1:5, fixedArgs = list(xdescr))

[[1]]
[1]     1 17770

[[2]]
```

```
[1]      1 480189

[[3]]
[1] 1 5

[[4]]
[1] 1999 2005

[[5]]
[1]  1 12
```

2.2.2 Shared memory via SNOW

In preparing *snow*, we used SSH keys to avoid having to enter a password for each of the workers. We chose to use sockets rather than MPI or PVM in this example (SNOW offers several choices for the underlying technology). The `stopCluster()` command may or may not be required, but is recommended by the authors of SNOW.

```
> library(snow)
> cl <- makeSOCKcluster(c("localhost", "localhost"))
> parSapply(cl, 1:5, worker, xdescr)

      [,1]   [,2] [,3] [,4] [,5]
[1,]     1      1    1 1999    1
[2,] 17770 480189    5 2005   12

> stopCluster(cl)
```

2.2.3 Interactive shared memory

Figure 1 shows two **R** sessions sharing the same copy of the Netflix data; this might be called "poor man's shared memory." The so-called "master" node (the left session) loads the data into shared memory and dumps the description to the file `netflixDescribe.txt`. It also calculates the mean ratings of two movies, "Against All Odds" (3.256693)and "Casino Royale" (2.897985). The worker session on the right attaches the matrix using the description in the file, and uses `head()` to show the success of the attachment. Next, the worker changes all the ratings of "Against All Odds" (movie 4943) to 100. Then both the worker and the master

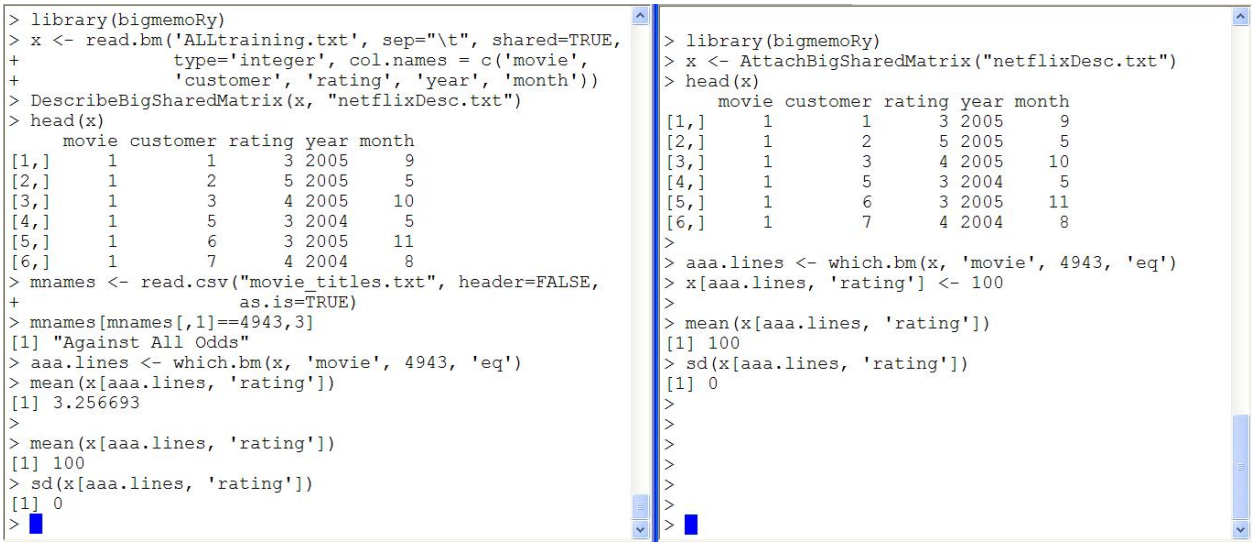

Figure 1: Sharing data across two R sessions using shared memory. The master session appears on the left, and the worker is on the right. The worker changes the ratings of movie 4943 ("Against All Odds"), and the change is reflected on the master via the shared matrix.

calculate the new mean (100) and standard deviation (0). The astute *nix programmer could easily do concurrent programming using shell scripts, `R CMD BATCH`, and `system()`, although this seems pointless given the ease of use of NetWorkSpaces and SNOW.

2.3 Parallel k-means with shared memory

Parallel k-means cluster analysis is not new, and others have proposed the use of shared memory (see (3), for example). The function `kmeans.bm()` supports the use of either NetWorkSpaces or SNOW for a parallel version of Lloyds's k-means algorithm (4) using shared memory. For sufficiently large and difficult problems, the speed improvement will be proportional to the number of processors. However, the most significant gains are in memory efficiency, where `kmeans.bm()` avoids the memory overhead of `kmeans()`.

The following example compares the parallel, shared-memory `kmeans.bm()` (using both NetWorkSpaces and SNOW) to **R**'s `kmeans()`; `kmeans.bm()` uses 4 workers, and the same data and the same starting points are used throughout.

```
> x <- BigSharedMatrix(3e+07, 5, init = 0, type = "double")
> x[seq(1, 3e+07, by = 3), ] <- rnorm(5e+07)
> x[seq(2, 3e+07, by = 3), ] <- rnorm(5e+07, 1, 1)
```

```
> x[seq(3, 3e+07, by = 3), ] <- rnorm(5e+07, -1, 1)
> centers <- x[1:3, ]
> start.bm.nws <- proc.time()[3]
> ans.nws <- kmeans.bm(x, centers, clients = 4)
> end.bm.nws <- proc.time()[3]
> ans.snow <- kmeans.bm(x, centers, clients = 4, parallel = "snow")
> end.bm.snow <- proc.time()[3]
> y <- x[, ]
> rm(x)
> gc(reset = TRUE)

            used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells    251326   13.5     467875   25.0    251326   13.5
Vcells 210317607 1604.6  551844356 4210.3 210317607 1604.6

> start.km <- proc.time()[3]
> ans.old <- kmeans(y, centers, algorithm = "Lloyd")
> end.km <- proc.time()[3]
> gc()

            used   (Mb) gc trigger (Mb)  max used   (Mb)
Ncells    252204   13.5     467875   25    259698   13.9
Vcells 225317814 1719.1  725078888 5532 720321568 5495.7
```

`kmeans()` requires 3 extra copies of the data, burning up about an extra 3.6 GB beyond the initial 1.2 GB data matrix. In contrast, `kmeans.bm` needs only two 240 MB vectors to manage the cluster memberships. In terms of speed of execution, the parallel versions are faster at completing the maximum of 10 iterations used here.

```
> end.bm.nws - start.bm.nws

elapsed
 28.345

> end.bm.snow - end.bm.nws

elapsed
 25.443
```

```
> end.km - start.km

elapsed
  38.61

> round(cbind(ans.nws$size, ans.nws$centers), 6)

         [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,]  9364087 -1.095498 -1.094226 -1.095403 -1.094580 -1.095341
[2,]  9869906  1.067050  1.066440  1.066267  1.066472  1.065635
[3,] 10766007 -0.025690 -0.025948 -0.024521 -0.026652 -0.023873

> round(cbind(ans.snow$size, ans.snow$centers), 6)

         [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,]  9364087 -1.095498 -1.094226 -1.095403 -1.094580 -1.095341
[2,]  9869906  1.067050  1.066440  1.066267  1.066472  1.065635
[3,] 10766007 -0.025690 -0.025948 -0.024521 -0.026652 -0.023873

> round(cbind(ans.old$size, ans.old$centers), 6)

      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
1  9364087 -1.095498 -1.094226 -1.095403 -1.094580 -1.095341
2  9869906  1.067050  1.066440  1.066267  1.066472  1.065635
3 10766007 -0.025690 -0.025948 -0.024521 -0.026652 -0.023873
```

Because none of the runs had converged at 10 interations, all were doing the same amount of work: the parallel version has a speed advantage in problems of this size. All the algorithms made the same progress, although slightly less progress than would have been made by MacQueen or Hartigan-Wong (2). The MacQueen algorithm updates the centers after every change in cluster membership, while the parallel algorithm (like the Lloyd algorithm) only updates the centers at the end of every pass through the data. We will improve `kmeans.bm()` in a future version of *bigmemoRy*: the copies of centers local to each process could be updated with local changes in cluster membership. For large problems, this would produce results closer to the pure algorithm of MacQueen. And when the algorithm is close to convergence, the shared memory global centers could be updated with minimal synchronization conflicts.

3 Architecture

We start by describing the features of *bigmemoRy* common to all platforms. We then discuss the use of shared memory which, as of Version 1.0, is only available for Unix-alikes (including Mac OS X). As with *ff*, we use long integers to index rows of the matrices in **C**; the size of a matrix is limited only by the memory limits of the operating system. There is one exception: on 32-bit systems, the address space may be exceeded if the user requests a vector (single-column matrix) of length greater than 2,147,483,647; this uses just over 2 GB of RAM (which may be available) but is not easily addressable.

3.1 Basic bigmemoRy design

We define an `S4` class `BigMatrix`, containing an external pointer (**R** type `externalptr`) and a type (a character string describing the type of the atomic element). The pointer gives the memory location of a **C** structure such as `BigIntMatrix` for an integer matrix (other types include double, short, and char).

```
> y <- BigMatrix(2, 2, type = "integer", init = 0)
> y
```

```
An object of class âĂIJBigMatrixâĂ
Slot "address":
<pointer: 0x15f508a0>

Slot "type":
[1] "integer"
```

```
> y[, ]
```

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

In **C**, the `BigIntMatrix` structure contains the number of rows and columns of the matrix, optional vectors of strings corresponding to row and column names, and a pointer to an array of pointers to integers (of length equal to the number of columns of the matrix).

The structure, including the massive vectors of data (one for each column of the matrix), is dynamically allocated using `malloc()`. We recognize that this is not the typical representation of a matrix as a massive vector, but it has some natural advantages for data exploration and analysis. Thus, the number of rows should greatly exceed the number of columns (and the user is advised to handle the transpose if this isn't the case, because of the overhead of pointer for each column).

Here, we show the structure of the `BigIntMatrix`, but we note that version 2.0 of *bigmemoRy* will change to **C++** classes and employ templates, creating somewhat more elegant code.

```
typedef struct
{
  long ncol;
  long nrow;
  int **matrix;
  ColumnNames *pColNames;
  RowNames *pRowNames;
  bool shared;
#ifndef WIN
  int *shcounter;
  int shcounterId;
  BMMutex shcounter_Mutex;
  ColumnMutexes *column_Mutexes;
  Ids *pColumnIds;
#endif
} BigIntMatrix;
```

We provide a collection of functions for the class `BigMatrix`, designed to mimic traditional matrix functionality. For example, the **R** function `ncol()` automatically associates a `BigMatrix` with a new method function, returning the number of columns of the matrix via the **C** function `CGetIntNcol()`. Again, we note that the organization of the **C** functions will change somewhat once we begin using templates. If `x` is a `BigMatrix` and `a` and `b` are traditional **R** vectors of row and column indices, then `x[a,b]` can be used to retrieve the appropriate submatrix of `x` (and is a traditional **R** matrix, not a `BigMatrix`). The same notation can be used to assign values to the submatrix, if the dimensionality of the assignment conforms to **R**'s set of permissible matrix assignments.

As discussed earlier, an object of class `BigMatrix` contains a *reference* to a data structure. This has some noteworthy consequences. First, if a `BigMatrix` is passed to a function which

changes its values, those changes will be reflected outside the function scope. Thus, *side-effects* have been introduced for `BigMatrix` objects. When an object of class `BigMatrix` is passed to a function, it should be thought of as being *called-by-reference*.

The second consequence is that copies are not made by simple assignment. For example:

```
> y[, ]
```

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

```
> z <- y
> z[1, 1] <- 1
> y[, ]
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    0
```

The assignment `z <- y` does not copy the contents of the `BigMatrix` object; it only copies the type information and address of the **C** structure. As a result, the **R** objects `z` and `y` refer to the same data in memory. If an actual copy is required, the function `deepcopy.bm()` should be used.

3.2 Shared memory design

We use the **C** system libraries `shm`, `ipc`, and `pthread` for shared memory management and mutual exclusions. The function `DescribeBigSharedMatrix` creates a description of the shared matrix that allows other processes to successfully attach the matrix. From the Netflix example, we have:

```
> xdescr
```

```
$type
[1] "integer"
```

```
$nrow
[1] 99072112

$rowNames
NULL

$ncol
[1] 5

$colNames
[1] "movie"    "customer" "rating"   "year"     "month"

$colKeys
[1] 234127365 234160134 234192903 234225672 234258441

$colMutexKeys
[1] 233963520 233996289 234029058 234061827 234094596

$shCountKey
[1] 234291210

$shCountMutexKey
[1] 234323979
```

We note the dimension and column names, discussed earlier. The last four components are keys which are used to identify locations in shared memory (for the matrix, a counter of the number of "attachments," and the mutual exclusions for read/write locks). The counter deserves special comment. The destructor is only allowed to destroy a shared matrix when there are no remaining processes attaching the matrix. So we keep the count of the number of attachments in shared memory (where it is updated by any attachment or detachment), and the actual destruction of the shared matrix occurs only when the last process (usually, but not necessarily, the master) is no longer attached to the matrix.

4 Conclusion

bigmemoRy supports double, integer, short, and char data types; in Unix environments, the package optionally implements the data structures in shared memory. Previously, parallel

use of **R** required redundant copies of data for each **R** process, and the shared memory version of a `BigMatrix` object now allows separate **R** processes on the same computer to share access to a single copy of the massive data set. *`bigmemoRy`* extends and augments the **R** statistical programming environment, opening the door for more powerful parallel analyses and data mining of massive data sets.

5 Extensions

Future work on *`bigmemoRy`* will hopefully include the addition of shared memory for Windows. We are also planning a redesign of the code using **C++** templates, and expanding the toolkit of file I/O functions.

References

[1] Daniel Adler, Oleg Nenadic, Walter Zucchini and Christian Glaeser (2007). ff: flat-file library. R package version 1.0-1, vignette http://cran.r-project.org/doc/vignettes/ff/ff.pdf.

[2] John Hartigan and M.A. Wong (1979). A K-means clustering algorithm. Applied Statistics 28, 100-108.

[3] Barbara Hohlt (2001). Pthread Parallel K-means. CS267 Applications of Parallel Computing, UC Berkeley, URL http://barbara.stattenfield.org/papers/cs267paper.pdf.

[4] S.P. Lloyd (1957). Lease squares quantization in PCM, Technical Note, Bell Laboratories. Published in 1982 in IEEE Transactions on Information Theory 28, 128-137.

[5] Thomas Lumley (2005). biglm: bounded memory linear and generalized linear models. R package version 0.4.

[6] J. MacQueen (1967). Some methods for clasification and analysis of multivariate observations. Proceedings of the 5^{th} Berkeley Symposium on Mathematical Statistics and Probability, UC Berkeley.

[7] The Netflix Prize competition. Netflix. http://www.netflixprize.com/

[8] REvolution Computing with support, contributions from Pfizer Inc. nws: R functions for NetWorkSpaces and Sleigh. R package version 1.6.2. http://nws-r.sourceforge.net/

[9] R Development Core Team (2007). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

[10] R Development Core Team (2007). R Installation and Administration. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-09-7, URL http://cran.r-project.org.

[11] Luke Tierney, A. J. Rossini, Na Li and H. Sevcikova. snow: Simple Network of Workstations. R package version 0.2-9.

[12] Herb Sutter (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3).

Appendix: Dangers of R's data frames and matrices

We love **R**'s data frames and matrices. The observations made here are not complaints, because there are excellent reasons behind choices made in the design and implementation of the programming environment. Here, we illustrate some dangers of **R**'s memory management. We will use a large matrix of integers as an example, motivated by the Netflix Prize competition.

Many **R** users are not aware that 4-byte integers are implemented in **R**. So if memory is at a premium and only integers are required, the astute and fastidious user would avoid the 2.24 GB memory consumption of

```
> x <- matrix(0, 1e+08, 3)
> round(object.size(x)/(1024^3), 2)
```

```
[1] 2.24
```

in favor of the 1.12 GB memory consumption of an integer matrix:

```
> x <- matrix(as.integer(0), 1e+08, 3)
> round(object.size(x)/(1024^3), 2)
```

```
[1] 1.12
```

Similar attention is needed in subsequent arithmetic, because

```
> x <- x + 1
```

coerces `x` into a matrix of 8-byte real numbers. The memory used is then back up to 2.24 GB, and the peak memory usage of this operation is approximately 3.91 GB (1.12 GB from the original `x`, and a new 2.24 GB vector). In contrast,

```
> x <- matrix(as.integer(0), 1e+08, 3)
> x <- x + as.integer(1)
```

has the desired consequence, adding 1 to every element of `x` while maintaining the integer type. The memory usage then remains at 1.12 GB, and the operation requires temporary

memory overhead of an additional 1.12 GB. It's tempting to be critical of *any* memory overhead for such a simple operation, but this is often necessary in a high-level, interactive programming environment. In fact, **R** is being efficient in the previous example, with peak memory consumption of 2.24 GB; the peak memory consumption of

```
> x <- matrix(as.integer(0), 1e+08, 3)
> x <- x + matrix(as.integer(1), 1e+08, 3)
```

is 3.91 GB. Some overhead is necessary in order to provide flexibility in the programming environment, freeing the user from the explicit structures of programming languages like **C**. However, this overhead becomes cumbersome with massive data sets.

Similar challenges arise in function calls, where **R** uses the *call-by-value* convention instead of *call-by-reference*. Fortunately, **R** creates physical copies of objects only when apparently necessary (called *copy-on-demand*). So there is no extra memory overhead for

```
> x <- matrix(as.integer(0), 1e+08, 3)
> myfunc1 <- function(z) return(c(z[1, 1], nrow(z), ncol(z)))
> myfunc1(x)

[1]         0 100000000         3
```

which uses only 1.12 GB for the matrix `x`, while

```
> x <- matrix(as.integer(0), 1e+08, 3)
> myfunc2 <- function(z) {
+     z[1, 1] <- as.integer(5)
+     return(c(z[1, 1], nrow(z), ncol(z)))
+ }
> myfunc2(x)

[1]         5 100000000         3
```

temporarily creates a second 1.12 GB matrix, for a total peak memory usage of 2.24 GB.

These examples demonstrate the simultaneous strengths and weaknesses of **R**. As a programming environment, it frees the user from the rigidity of a programming language like **C**. The resulting power and flexibility has a cost: data structures can be inefficient, and even the simplest manipulations can create unexpected memory overhead. For most day-to-day use, the strengths far outweigh the weaknesses. But when working with massive data sets, the even the best efforts can grind to a halt.