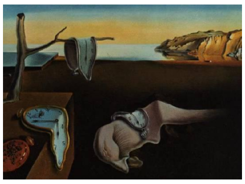

Journal of Statistical Software

November 2013, Volume 55, Issue 14. http://www.jstatsoft.org/

Scalable Strategies for Computing with Massive Data

Michael J. Kane
Yale University

John W. Emerson
Yale University

Stephen Weston
Yale University

Abstract

This paper presents two complementary statistical computing frameworks that address challenges in parallel processing and the analysis of massive data. First, the **foreach** package allows users of the R programming environment to define parallel loops that may be run sequentially on a single machine, in parallel on a symmetric multiprocessing (SMP) machine, or in cluster environments without platform-specific code. Second, the **bigmemory** package implements memory- and file-mapped data structures that provide (a) access to arbitrarily large data while retaining a look and feel that is familiar to R users and (b) data structures that are shared across processor cores in order to support efficient parallel computing techniques. Although these packages may be used independently, this paper shows how they can be used in combination to address challenges that have effectively been beyond the reach of researchers who lack specialized software development skills or expensive hardware.

Keywords: concurrent programming, memory-mapping, parallel programming, shared memory, statistical computing.

1. Introduction

The analysis of increasingly large data sets and the use of parallel processing are active areas of research in statistics and machine learning. Examples of these data sets include the Nexflix Prize competition (Bennet and Lanning 2007); next-generation genome sequencing and analysis; and the American Statistical Association's 2009 Data Expo involving the airline on-time performance data set (RITA 2009). Many statisticians are left behind when confronted with massive data challenges because their two most widely-used software packages, SAS (SAS Institute Inc. 2011) and R (R Core Team 2013a), are ill-equipped to handle this class of problem. SAS supports the analysis of large data with an impressive number of standard methods, but the Netflix Prize competition required the development and implementation of new methodologies. On the other hand, R is very well-suited for the development of new data analysis and statistical techniques but does not seamlessly handle massive data sets. These barriers to entry have presented significant obstacles to statisticians interested in engaging in such massive data challenges.

The root of the problem is the current inability of modern high-level programming environments like R to exploit specialized computing capabilities. Package **bigmemory** (Kane, Emerson, and Haverty 2013a) leverages low-level operating system features to provide data structures capable of supporting massive data, potentially larger than random access memory (RAM). Unlike database solutions and other alternative approaches, the data structures provided by **bigmemory** are compatible with standard basic linear algebra subroutines (**BLAS**), linear algebra package (**LAPACK**) subroutines, and any algorithms which rely upon column-major matrices. The data structures are available in shared memory for use by multiple processes in parallel programming applications and can be shared across machines using supported cluster filesystems. The design of **bigmemory** addresses two interrelated challenges in computing with massive data: data management and statistical analysis. Section 2 describes these challenges further and presents solutions for managing and exploring massive data. When the calculations required by an exploration or analysis become overwhelming parallel computing techniques can be used to decrease execution time. Package **foreach** (Revolution Analytics and Weston 2013b) provides a general, technology-agnostic framework for implementing parallel algorithms and can exploit the shared memory capabilities of **bigmemory**. Section 4 considers a broad class of statistical analyses well-suited to **foreach** parallel computing capababilities. Thus, **bigmemory** and **foreach** can be used together to provide a software framework for computing with massive data (demonstrated in Section 3) that includes shared memory, parallel computing capabilities (demonstrated in Section 5). Section 6 examines the performance of **bigmemory** and **foreach** compared to standard R data structures and parallel programming capabilities in a small data setting. Section 7 concludes with a discussion of the future of massive data and parallel computing in the R programming environment.

2. Big data challenges and bigmemory

High-level programming environments such as R and MATLAB (The MathWorks, Inc. 2011) allow statisticians to easily import, visualize, manipulate, and model data as well as develop new techniques. However, this convenience comes at a cost because even simple analyses can incur significant memory overhead. Lower-level programming languages sacrifice this convenience but often can reduce the overhead by *referencing* existing data structures instead of creating unnecessary temporary copies. When the data are small, the overhead of creating copies in high-level environments is negligible and generally goes unnoticed. However, as data grow in size the memory overhead can become prohibitively expensive. Programming environments like R and MATLAB have some referencing capabilities, but their existing functions generally do not take advantage of them. Uninformed use of these features can lead to unanticipated results.

According to the R Installation and Administration manual (R Core Team 2013b), R is not well-suited for working with data structures larger than about 10–20% of a computer's RAM. Data exceeding 50% of available RAM are essentially unusable because the overhead of all but the simplest of calculations quickly consumes all available RAM. Based on these guidelines, we consider a data set *large* if it exceeds 20% of the RAM on a given machine and *massive* if it exceeds 50%. Although the notion of size varies with hardware capability, the challenges and solutions of computing with massive data scale to the statistician's problem of interest and computing resources. We are particular interested in cases where massive data or the memory overhead of an analysis exceed the limits of available RAM. In such cases, computing requires use of fixed storage (disk space) in combination with RAM.

Historically, size limitations of high-level programming languages have resulted in the use of a database. A database can provide convenient access to subsets of large and massive data structures and are particularly efficient for certain types of summaries and tabulations. However, reliance on a database has several drawbacks. First, it can be relatively slow in executing many of the numerical calculations upon which statistical algorithms rely. Second, calculations not supported by the database require copying subsets of the data into objects of the high-level language stored in RAM. This copying can be slow and the subsequent analysis may require customized implementations of algorithms for cases where the overhead of standard implementations (used for smaller data) exceeds the capacity of RAM. Finally, the use of a database requires the installation and maintenance of a software package separate from the high-level programming environment and may not be available to all researchers.

Customized extraction of subsets of data from files resident on disk provides an alternative to databases. However this approach suffers from two drawbacks. First, it has to be done manually, requiring extra time to develop specialized code and, as a result, proportionally more time for debugging. Second, custom extractions are often coupled to specific calculations and cannot be implemented as part of a general solution. For example, the calculation of the mean of a column of a matrix requires an extraction scheme that loads only elements from the column of interest. However, a column-wise extraction will not work well for calculating matrix row means. Furthermore, the modes of extraction are specific to the chosen file format and data structures. As a result, different extraction schemes may need to be implemented over the course of a data exploration depending on how the data are stored. This may further increase development and debugging time. In the extreme case, some calculations may require sophisticated extraction schemes that may be prohibitively difficult to implement; in such cases the statistician is effectively precluded from performing these types of calculations.

Both the database and custom extraction approaches are limited because their data structures on disk are not numerical objects that can be used directly in the implementation of a statistical analysis in the high-level language. They require loading small portions of the data from disk into data structures of the high-level language in RAM, completing some partial analysis, and then moving on to other subsets of the data. As a result, existing code designed to work on entire data structures native to the language and within RAM is generally incompatible with analyses relying upon database or customized extractions of data from files. Fundamentally, these approaches are limited by their reliance on technologies that evolved decades ago and lack the flexibility provided by modern computing systems and languages. Some algorithms have been designed specifically for use with massive data. For example, the R package **biglm** (Lumley 2013) implements an incremental algorithm for linear regression (Miller 1992) that processes the data in chunks, avoiding the memory overhead of R's native `lm` function for fitting linear models. However, such solutions are not always possible. The prospect of implementing different algorithms for a certain type of analysis simply to support different data sizes seems grossly inefficient.

2.1. Underlying technology

Modern operating systems allow files resident on disk to be *memory-mapped*, associating a segment of virtual memory in a one-to-one correspondence with contents of a file. The `C` function `mmap` is available for this memory-mapping on POSIX-compliant operating systems (including UNIX, Linux, and Mac OS X, in particular); Microsoft Windows offers similar functionality. The **Boost** `C++` Libraries (Dawes *et al.* 2013) provide an application programming interface for the use of memory-mapped files which allows portable solutions across Microsoft Windows and POSIX-compliant systems (made available through the **BH** package Emerson, Kane, Eddelbuettel, Allaire, and Francois 2013). Memory-mapped files provide faster access to data than standard read and write operations for a variety of reasons beyond the scope of this paper. Most importantly, the task of moving data between disk and RAM (called *caching*) is handled at the operating-system level and avoids the inevitable costs associated with an intermediary such as a database or a customized extraction solution. Interested readers should consult one of the many web pages describing memory-mapping, such as Kath (1993) for example.

Memory-mapping is the cornerstone of a scalable solution for working with massive data. Size limitations become associated with available file resources (disk space) rather than RAM. From the perspective of both the developer and end-user, only one type of data structure is needed to support data sets of all sizes, from miniscule to massive. The resulting programming environment is thus both efficient and scalable, allowing single implementations of algorithms for statistical analyses to be used regardless of the size of the problem. When an algorithm requires data not yet analyzed, the operating system automatically caches the data in RAM. This caching happens at a speed faster than any general data management alternative and only slower than customized solutions designed for very specific purposes. Once in RAM, calculations proceed at standard in-memory speeds. Once memory is exhausted, the operating system handles caching of new data and displacing older data (which are written to disk if modified). The end-user is insulated from the details of this mechanism, which is certainly not the case with either database or customized extraction approaches.

2.2. The bigmemory family of packages

We offer a family of packages for the `R` statistical programming environment for computing with massive data for POSIX-compliant operating systems. Windows is currently not supported but could be through the POSIX-compatible environments used by the `R` environment. This family of packages is intended for data that can be represented as a matrix and on computers with 64-bit operating systems. The main contribution is a new data structure providing a dense, numeric matrix called a `big.matrix` which exploits memory-mapping for several purposes. First, the use of a memory-mapped file (called a *filebacking*) allows matrices to exceed available RAM in size, up to the limitations of available file system resources. Second, the matrices support the use of shared memory for efficiencies in parallel computing. A `big.matrix` can be created on any file system that supports `mmap`, including cluster file systems. As a result, **bigmemory** is an option for large-scale statistical computing, both on single machines or on a cluster of machines with the appropriate configuration. The support for shared-memory matrices and a new framework for portable parallel programming will be discussed in Section 4. Third, the data structure provides reference behavior, helping to avoid the creation of unnecessary temporary copies of massive objects. Finally, the underlying matrix data structure is in standard column-major format and is thus compatible with existing **BLAS** and **LAPACK** libraries as well as other legacy code for statistical analysis (primarily implemented in C, C++, and Fortran).

These packages are immediately useful in R, supporting the basic exploration and analysis of massive data in a manner that is accessible to non-expert R users. Typical uses involve the extraction of subsets of data into native R objects in RAM rather than the application of existing functions (such as `lm`) for analysis of the entire data set. Some of these features can be used independently of R by expert developers in other environments having a C++ interface. Although existing algorithms could be modified specifically for use with `big.matrix` objects, this opens a Pandora's box of recoding which is not a long-term solution for scalable statistical analyses. Instead, we support the position taken by Ihaka and Temple Lang (2008) in planning for the next-generation statistical programming environment (likely a new major release of R). At the most basic level, a new environment could provide seamless scalability through filebacked memory-mapping for large data objects within the native memory allocator. This would help avoid the need for specialized tools for managing massive data, allowing statisticians and developers to focus on new methodology and algorithms for analyses.

The matrices provided by **bigmemory** offer several advantages over standard R matrices, but these advantages come with tradeoffs. Here, we highlight the two most important qualifications. First, a `big.matrix` can rarely be used directly with existing R functions. A `big.matrix` has its own class and is nothing more than a reference to a data structure which can't be used by functions such as `base::summary`, for example. However, many analogous `big.matrix` operations are included in package **biganalytics** (Emerson and Kane 2013a) which implements `apply`, `biglm`, `bigglm`, `bigkmeans`, `colmax`, `colmin`, `colmean`, `colprod`, `colrange`, `colvar`, `summary`, etc. Tabulation operations are included in package **bigtabulate** (Kane and Emerson 2013b) and includes functions `bigsplit`, `bigtabulate`, `bigtable`, and `bigtsummary`. Other analyses will need to be conducted on subsets of the `big.matrix` (which must fit into available RAM as R matrices). Similarly, the extraction of a larger-than-RAM subset of a `big.matrix` into a new `big.matrix` must be done manually by the user, a simple two-step process of creating the new object and then conducting the copy in chunks. The one exception to this is the `sub.big.matrix` class, which creates "windows" into contiguous, rectangular blocks for a `big.matrix` and is beyond the scope of this article. Second, **bigmemory** supports numeric matrices (including `NA` values) but not character strings or anything like a `big.data.frame`. Package **ff** (Adler, Gläser, Nenadic, Oehlschlägel, and Zucchini 2013) offers a wide range of advanced functionality but at the cost of **BLAS** and **LAPACK** compatibility; a full comparison of **bigmemory** and **ff** is beyond the scope of this paper.

3. Application: Airline data management

Data analysis usually begins with the importation of data into native data structures of a statistical computing environment. In R, this is generally accomplished with functions such as `read.table`, which is very flexible. It reads a text file into a `data.frame` and it can perform this operation without information regarding the column types. To remain robust and to correctly read these files there are many checks that need to be performed while a file is being imported and there is associated overhead with this process. This is particularly true when the `colClasses` parameter is not specified and the `read.table` function is required to derive the column type while scanning through the rows of a data file. In this case, an intermediate `data.frame` is created with intermediate column vectors. If the data in a subsequent scan corresponds to a different type than an intermediate vector, then a new vector is created, with the updated type, and the intermediate vector is copied to the new vector. The process continues until all rows of the data file are scanned. As a result, these importing functions can incur significant memory overhead resulting in long load times for large data sets. Furthermore, native R data structures are limited to RAM. Together, these limitations have made it difficult, or even precluded, many statisticians from exploring large data using R's native data structures.

The **bigmemory** package offers a useful alternative for massive numeric data matrices. The following code creates a filebacking for the airline on-time performance data.

This data set contains 29 records on each of approximately 120 million commercial domestic flights in the United States from October 1987 to April 2008, and consumes approximately 12 GB of memory. A script to convert the data set into integer values is available at the Bigmemory Project website (Emerson and Kane 2013b). A compressed version of the preprocessed data set along with all of the examples that appear in this article can be downloaded from the author's website (Kane and Emerson 2013a). The reader should note that the compressed file is approximately 1.7 GB and requires approximately 12 GB. The creation of the filebacking avoids significant memory overhead and, as discussed in Section 2, the resulting `big.matrix` may be larger than available RAM. Double-precision floating point values are used for indexing, so a `big.matrix` may contain up to $2^{53} - 1$ elements for 64-bit architectures, such as x86_64 (1 petabyte of 4-byte integer data, for example). These values are type-cast to 64-bit integers. This approach will be used for all numerical, vector-based data structures as of R version 3.0.0, available beginning of April 2013. The creation of the filebacking for the airline on-time performance data takes about 15 minutes. In contrast, the use of R's native `read.csv` would require about 32 GB of RAM, beyond the resources of common hardware circa 2013. The creation of an **SQLite** database **SQLite** Development Team (2013) takes more than an hour and requires the availability and use of separate database software.[1]

At the most basic level, a `big.matrix` object is used in exactly the same way as a native R `matrix` via the bracket operators (`"["` and `"[<-"`) for extractions and assignments. One notable difference is that the filebacking only needs to be created once, entirely avoiding the need for the further use of `read.csv` or `read.big.matrix`. The following code creates a filebacking.

```
R> library("bigmemory")
R> x <- read.big.matrix("airline.csv", header = TRUE,
+    backingfile = "airline.bin", descriptorfile = "airline.desc",
+    type = "integer")
R> dim(x)

[1] 123534969        29
```

A subsequent R session may instantly reference, or attach to, this filebacking.

```
R> library("bigmemory")
R> x <- attach.big.matrix("airline.desc")
```

---

[1] These benchmarks were performed on a machine running Ubuntu 12.04 (64-bit), 18 GB of RAM, and Intel Core i7 CPU x 980 @ 3.33 GHz with 6 processor cores.

```
R> dim(x)
```

```
[1] 123534969        29
```

Subsets of data may be extracted and used with native R functions. In the following code, for example, `x[, "DepDelay"]` creates an R vector of integer departure delays of length 123,534,969, consuming about 600 megabytes (MB). This operation is handled on most modern hardware without difficulty, leaving the researcher working with familiar commands on such extracted subsets of data. As an aside, it may be noted that the minimum departure delay of −1410 minutes deserves careful attention and perhaps data cleaning.

```
R> summary(x[, "DepDelay"])
```

```
       Min.      1st Qu.       Median         Mean 
  -1410.000       -2.000        0.000        8.171 
    3rd Qu.         Max.         NA's 
      6.000       2601.0    2302136.0 
```

A second example illustrates some important features relating to the performance of the underlying `mmap` technology. Consider the task of first calculating the minimum value of a variable, `year`, followed by the extraction of that same column from the data set. R can ask a database to return the minimum value of `year` from an airline database (called `airline`): `from_db("select min(year) from airline")`. This takes about two minutes. If, immediately afterwards, the entire `year` column is extracted into an R vector, the database query `a <- from_db("select year from airline")` also takes about two minutes. In contrast, determining the minimum value of `year` using `colmin(x, "year")` takes about eight seconds and a subsequent extraction, `b <- x[, "year"]`, takes a mere two seconds. This example illustrates two important points about the memory management solution implemented in **bigmemory**. First, the low-level caching performance of **bigmemory** is an order of magnitude better than the database. Second, the caching benefits from an awareness of the presence of the `year` data in RAM following the calculation of the column minimum. The subsequent extraction takes place entirely in RAM without any disk access.

4. Flexible parallel programming with foreach

When confronted with a fascinating real-data problem, statisticians quickly move beyond basic summaries to more sophisticated explorations and formal analyses. Many such explorations involve repeated calculations on subsets of data. This section presents a flexible, easy-to-use parallel framework for this class of computing challenges. This framework is useful for many typical applications and also has critical advantages when working with massive data. Section 5 provides examples illustrating the points developed here.

We begin by considering the general problem of conducting repeated calculations on subsets of data, common in statistical explorations and analyses. Many analyses can be accomplished by grouping data (called the *split*), performing a single calculation on each group (the *apply*), and returning the results in a specified format (the *combine*). The term “split-apply-combine” was coined by Wickham (2011) but the approach has been supported on a number of different computing environments for some time under different names. The approach was originally implemented in the mid 1980's with the GAMMA (DeWitt, Gerber, Graefe, Heytens, Kumar, and Muralikrishna 1986) and Bubba (Boral, Alexander, Clay, Copeland, Danforth, Franklin, Hart, Smith, and Valduriez 1990) databases. SAS implements the approach through its `by` statement. Apache's **Hadoop** software packages (The Apache Software Foundation 2013) implement split-apply-combine operations for distributed systems. The design for a split-apply-combine framework has also been explored for relational databases as described in Chen, Therber, Hsu, Zeller, Zhang, and Wu (2009). Each of these implementations attempts to simultaneously support data management and analysis in a framework that lends itself to parallel computing. However, most other computing environments lack R's extensive statistical capabilities, and other environments' support for concurrent programming is neither portable nor easy to exploit.

There are several benefits to the split-apply-combine approach that may not be obvious. First, split-apply-combine is computationally efficient. It only requires two passes through the data: one to create the groups and one to perform the analysis. Admittedly, storing the groups from the split step requires extra memory but this overhead is usually manageable. In contrast, a naive approach makes a costly pass through the data for each group, adding an order of magnitude to the computational complexity of the analysis. Second, the apply step is an ideal candidate for parallel computing. When the calculation is intensive, parallelizing the apply step can dramatically reduce the execution time.

Although the split-apply-combine approach provides an attractive opportunity for parallel computing, the parallelization of algorithms has historically been cumbersome and suffers from two serious drawbacks. First, it requires that the statistician re-implements existing code, repeating the process of development, testing, and debugging. Second, there are a plethora of different parallel mechanisms, each with its own unique interface. For R, examples include **multicore** (Urbanek 2011), **snow** (Tierney, Rossini, Li, and Sevcikova 2013), **parallel** (R Core Team 2013a), and **Rmpi** (Yu 2013, 2002). As a result, a different version of the parallel code is needed for each of the parallel mechanism.

We introduce the **foreach** package to solve both of these historic difficulties with parallel programming. The package is not a new parallel mechanism, but a new framework for parallel programming that makes use of existing parallel mechanisms. The goal is to allow the statistician to create concurrent loops which can be used independently of the choice of a particular parallel mechanism. A `foreach` loop can be run sequentially, in parallel on a single machine, or in parallel on a cluster of machines without requiring any re-implementation or code modification of the algorithm itself. Figure 1 illustrates this point. The example loads the packages **foreach** and **doSNOW** (Revolution Analytics 2013b), allowing the `foreach` loop to use the **snow** parallel mechanism. A "cluster" of two workers on the local machine is created with the `makeCluster` function. The `foreach` function is made aware that **snow** should be used for parallel computation with the `registerDoSNOW` function call. Loading and registering the parallel mechanism requires no changes to the algorithm, designated as the "real work." The algorithm itself assumes that `G` represents a partition of the row indices of the data `X` into groups. The `foreach` loop iterates through each of the groups in `G`. The `%dopar%` binary operator specifies that each iteration of the loop should be run in parallel using the registered mechanism. The body of the loop specifies the code that is run on the parallel processes. In this case `f` is some function implementing an analysis, and `Y` represents (optional) auxiliary data.

```
# Optional declaration of a parallel mechanism here, such as
# the following for a 2-worker SNOW parallel environment:
library("foreach")
library("doSNOW")
cl <- makeCluster(2)
registerDoSNOW(cl)

# The real work:
ans <- foreach(g = G) %dopar% {
  f(X[g,], Y)
}
```

Figure 1: The general `foreach` framework.

The **snow** package is not the only mechanism compatible with **foreach**; **foreach** provides a framework for supporting new or alternative parallel mechanisms. There are currently five open-source packages that provide compatibility between various parallel environments and **foreach**: **doMC** (Revolution Analytics 2013a), **doMPI** (Weston 2013), **doRedis** (Lewis 2012), **doParallel** (Revolution Analytics and Weston 2013a), and **doSNOW**. These implementations are freely available for use with the associated parallel environments.

Parallelizing the apply step comes with its own challenges because there is often contention between data size and the number of parallel processes that can be utilized. If the amount of data required by a parallel process is large, then extra time is usually required to copy the data to the worker. In addition, data copies received by the worker consume valuable RAM. This limits the number of parallel workers that can be employed on a machine as data copies eventually consume available RAM. It should be noted that **multicore** can often avoid this copying overhead, a point that will be addressed later in the paper.

We illustrate these challenges with a simple example using the following code:

```
R> require("foreach")
R> x <- read.csv("1995.csv", as.is = TRUE)
R> groups <- split(1:nrow(x), x[, "DayOfWeek"])
R> depDelayQuantiles <- foreach(g = groups, .combine = rbind) %dopar% {
+      quantile(x[g, "DepDelay"], probs = c(0.5, 0.9, 0.99),
+        na.rm = TRUE)
+  }
R> rownames(depDelayQuantiles) <- c("Mon", "Tues", "Wed", "Thu", "Fri",
+    "Sat", "Sun")
```

We consider the calculation of 50%, 90%, and 99% quantiles of departure delays for each day of the week for only the 1995 flights in the airline data set. Although this example only requires two of the 29 variables and is thus somewhat contrived, many big-data problems would make use of the complete data and the principle illustrated here provides a scalable solution. We use the split-apply-combine approach and examine the consequences of parallel programming. After reading in the airline data for the year 1995, the row indices are split by the day of the week of each flight. The resulting `groups` variable is a named list with names corresponding to the day of the week and the vectors denote the rows in the data where a flight occurred for a given day. The elements of the resulting `groups` variable are iterated over in the `foreach` loop. The body of the loop computes the quantiles for the departure delays for the day of the week specified by `g`. The result of the body of the loop is a vector of quantile values that are combined with the `rbind` function, resulting in a matrix, that is stored as the `depDelayQuantiles` variable. For convenience and ease-of-use, the rows of `depDelayQuantiles` are named according to their respective days.

Over two GB of RAM is required when performing this calculation using four parallel processes. If this exhausts available RAM, the operating system would try to compensate by "swapping" inactively-used RAM to disk. However, swapping is inefficient and results in much longer execution times. Thus, the memory overhead of a parallel worker effectively dictates the total number of parallel processes that can be employed.

5. Combining bigmemory and foreach

The challenges described in the previous section can be effectively addressed by providing shared-memory data structures. Memory overhead associated with copying data to worker processes can be eliminated when shared-memory data structures are referenced by multiple R sessions. Instead of creating a copy, a parallel worker receives a descriptor of a `big.matrix` which allows for immediate access to the data via shared memory. Hence, the framework implemented by **bigmemory** and **foreach** dramatically increases the ease of development and efficiency of execution (both in terms of speed and memory consumption) for parallel problems working on massive sets of data.

We present a detailed example of the split-apply-combine approach to finding quantiles of flight delays using all years of the airline data set. Even if the entire data set were available as a 12 GB native R matrix in RAM (beyond typical capabilities in 2012), the memory overhead of providing copies of the data to parallel processes would be prohibitively expensive. Instead, we provide a scalable parallel solution making use of shared memory. The "worker" function calculates the desired quantiles for a subset of data given by row indices `rows` and is defined by:

```
R> GetDepQuantiles <- function(rows, data) {
+    quantile(data[rows, "DepDelay"], probs = c(0.5, 0.9, 0.99),
+    na.rm = TRUE)
+  }
```

The sequence of steps in this scalable solution are given in the following lines of code.

First, the filebacked airline data is attached (as described in Section 2 and illustrated in Section 3).

```
R> library("bigmemory")
R> x <- attach.big.matrix("airline.desc")
```

Second, a parallel environment is defined (as described in Section 4). This step could be omitted, in which case the calculations would be executed sequentially on a single processor.

Quantile	Mon	Tues	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

Table 1: Quantiles of departure delays by day.

The following therefore is an optional declaration of a parallel mechanism, where we specify a 4-worker SNOW parallel environment:

```
R> library("foreach")
R> library("doSNOW")
R> cl <- makeSOCKcluster(rep("localhost", 4))
R> registerDoSNOW(cl)
```

Next, a "descriptor" of the shared-memory airline data, x, is obtained.

```
R> xdesc <- describe(x)
```

Third, the split step of the split-apply-combine approach divides the row indices into seven groups, one for each day of the week (as in Section 4).

```
R> G <- split(1:nrow(x), x[, "DayOfWeek"])
```

Finally, `foreach` is used to define the calculations required of each worker process, i.e., the real work is done. Access to the shared-memory airline data is provided by attaching itand the departure quantiles for the specific day are obtained using `GetDepQuantiles()`.

```
R> qs <- foreach(g = G, .combine = rbind) %dopar% {
+    require("bigmemory")
+    x <- attach.big.matrix(xdesc)
+    GetDepQuantiles(g, x)
+  }
R> rownames(qs) <- c("Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun")
```

The two important characteristics of this solution are (a) only the descriptor is passed to the parallel worker, not the large data set and (b) neither the worker function `GetDepQuantiles` nor the body of the `foreach` loop requires modification for sequential computation or use with alternative parallel computing environments. The result of this toy exploration appears in Table 1, showing that Tuesdays and Saturdays had less severe departure delays.

6. Benchmarks

When used together, **bigmemory** and **foreach** provide a scalable framework that allows for the exploration of massive sets of data. However, it is important to note that they also provide benefits when used with smaller data. In this case, their performance is competitive with R's built-in data structures, like `matrix`, and parallel programming capabilities available in the base package **parallel**. To better understand these benefits and to benchmark[2] performance, we constructed square matrices with elements sampled from the uniform distribution. We selected random subsets of half of the columns. For each of these columns we calculated sample medians of 10 bootstrap samples. In the first benchmark, we considered different sized matrices, having between 5,000 and 11,180 ($\sim \sqrt{1.25 \times 10^8}$) rows and columns, holding fixed the number of processor cores at four. Fixing the degree of parallelization in this way allows us to examine how each approach scales with the data size. In the second benchmark, we fixed the matrix size to be 7,071 ($\sim \sqrt{5 \times 10^7}$) rows and columns, and examined performance using between 2 and 6 processor cores. Fixing the matrix size in this way allows us to examine how each approach scales in the number of parallel processes. The following code gives the implementation using `mclapply`; minor modifications are needed for the sequential and `parLapply` implementations.

```
R> a <- matrix(runif(5000 * 5000), nrow = 5000, ncol = 5000)
R> sel <- sample(nrow(a), floor(nrow(a)/2))
R> bootstrapEstimate <- mclapply(X = sel, mc.cores = 4,
+    FUN = function(x) {
+      bootstrapMedians <- rep(NA, 10)
+        for (j in 1:10) {
+          bootstrapMedians[j] <-
+            as.numeric(quantile(sample(a[, x],
+              nrow(a), replace = TRUE), probs = 0.5))
+        }
+        c(mean(bootstrapMedians), sd(bootstrapMedians))
+    })
```

The implementation using **foreach** and **bigmemory** is given by:

```
R> require("doMC")
R> registerDoMC(cores = 4)
R> y <- as.big.matrix(a, backingfile = "benchmark.back",
+    descriptorfile = "benchmark.desc")
R> ydesc <- describe(y)
R> bootstrapEstimate <- foreach(x = sel) %dopar% {
+    require("bigmemory")
+    y <- attach.big.matrix(ydesc)
+    bootstrapMedians <- rep(NA, 10)
+    for (j in 1:10) {
+      bootstrapMedians[j] <- as.numeric(quantile(sample(y[, x],
+        nrow(y), replace = TRUE), probs = 0.5))
+    }
+    c(mean(bootstrapMedians), sd(bootstrapMedians))
+  }
```

---

[2] These benchmarks were performed on a machine running Ubuntu 12.04 (64-bit), 18 GB of RAM, and Intel Core i7 CPU x 980 @ 3.33 GHz with 6 processor cores.

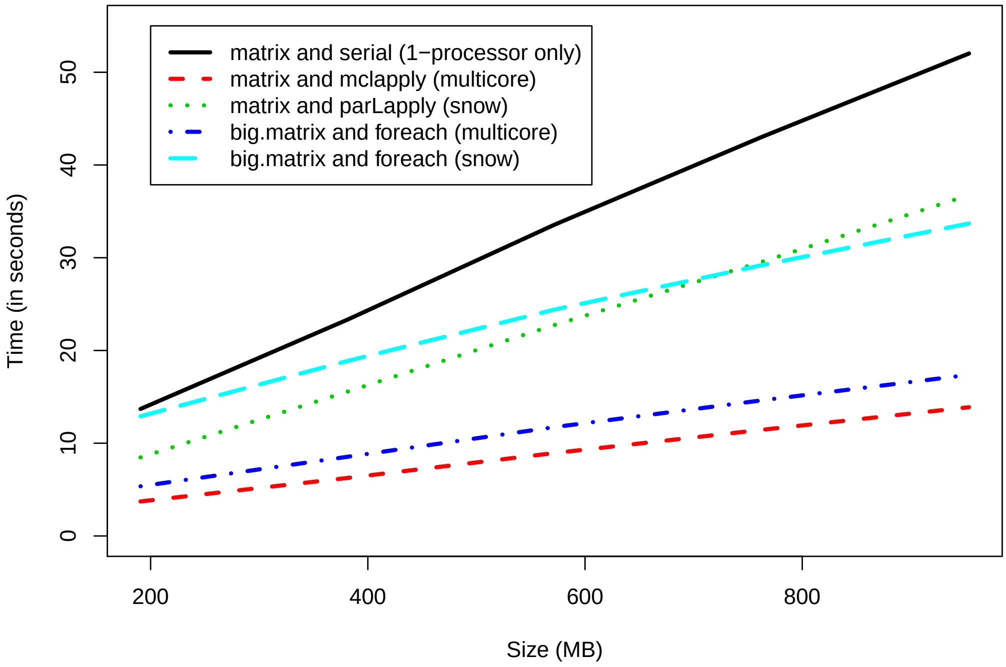

Figure 2: The benchmark results for varying simulation sizes using four processor cores.

Out of the combinations presented only **bigmemory** and **foreach** can scale past the size of available RAM; of the others, **snow** incurs the most overhead since copies of the data must be made for each parallel process. However, this overhead can be mitigated when matrix data is managed using **bigmemory** since parallel workers can receive descriptors, and then attach to a `big.matrix` object, rather than transmitting the data for an entire matrix.

Figure 2 shows the benchmark times for a range of simulation sizes using four processor cores. The sequential timing (labelled "serial") is included for reference. In all cases the timing increases linearly with the data size. The slope associated with the `parLapply` function, which makes use of the **snow** package, is approximately two times steeper because of the overhead of sending copies of the data to workers. This overhead is mitigated in the case of **snow** and **bigmemory** since matrices do not need to be transmitted to parallel processes. The descriptor is transmitted and the `big.matrix` is attached, using fewer memory resources. The `mclapply` function, which makes use of **multicore**, gives the greatest speed gains. However, the approach using `foreach` with **multicore** and shared-memory `big.matrix` objects is only slightly slower than `mclapply` with R's native matrices.

Figure 3 shows the benchmark results for a fixed matrix size, considering performance as the number of parallel processes ranges from two to six. Again, the sequential timing is included and labelled "serial" for reference. Gains in performance are seen with `parLapply` (which uses **snow**) only up to four cores. When more than four cores are employed the communication overhead required by **snow** overwhelms the performance gains. When five cores are employed the fixed cost of using **foreach** is less than the price of transmitting R matrices to parallel processes and the combination of **foreach** and **bigmemory** sees better performance than its **`matrix-snow`** analogue. The packages relying on **multicore** see initial decreases in timing but then the improvements decline as the overhead becomes comparable to the gains seen

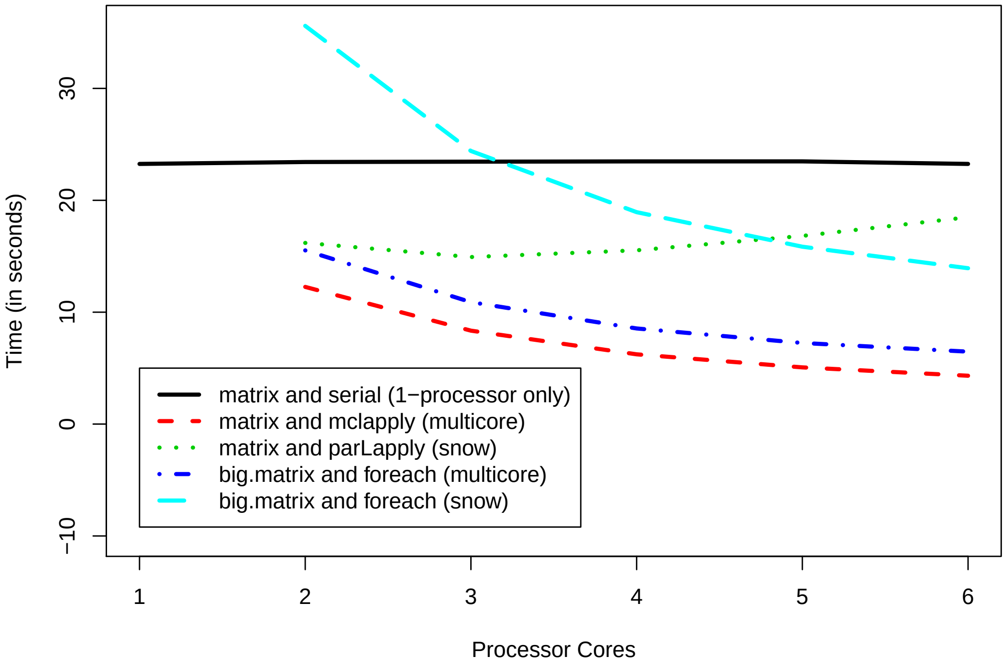

Figure 3: The benchmark results for a fixed matrix size using different numbers of processor cores.

from the use of additional parallel processes. As in Figure 2 it is clear that the performance of **bigmemory** and **foreach** is comparable to R's native matrices used in conjunction with **multicore**.

It is important to understand that the parallel packages **multicore** and **snow** incur very different types of overhead resulting in distinct profiles as they are benchmarked on differently sized data. The **multicore** package creates parallel workers by "forking" an R process. This creates a copy of the entire R process where memory in the new process is only copied before it is modified. This fork operation and copy-on-write behavior is quite efficient. The **snow** package takes a different approach, spawning worker processes separately from the calling R process. During a parallel computation all data needed are copied to worker processes. Thus, **snow** generally incurs greater communication overhead but has the advantage of working on Windows and Unix platforms. However, **snow**'s use is not limited to a single machine; it can be used for parallel computing on a cluster of machines for demanding, large-scale calculations. For either of these packages, and for parallel computing in general, it is important to understand the associated overhead and make sure that individual computations are substantial enough to justify the use of parallel computing techniques.

7. Discussion

This paper presents a range of contributions in computing with massive data, shared memory, and parallel programming. It focuses on the R packages **bigmemory** and **foreach**. Other packages provide additional features. Package **biganalytics** (Emerson and Kane 2013a) offers k-means cluster analysis as well as linear and generalized linear model support for use with the massive data structures. Package **synchronicity** (Kane 2013) supports process synchronization necessary for advanced parallel programming challenges. For example, package **NMF** (Gaujoux 2013; Gaujoux and Seoighe 2010) relies upon **synchronicity** for conducting a non-negative matrix factorization with a parallel algorithm requiring a sophisticated synchronization scheme. Package **bigalgebra** (Kane, Lewis, and Emerson 2013b) allows native R matrices and `big.matrix` objects to be sent to optimized, 64-bit **BLAS** and **LAPACK** libraries and can be used in conjunction with the partial singular value decomposition as described in Baglama and Reichel (2005) and implemented in package **irlba** (Baglama and Reichel 2012).

Initial massive data explorations often lead to formal analyses using well-established methodology, but they could also lead to the development of new methodology. Some existing methods may be easily re-implemented with filebacked memory-mappings for use with massive data. Other methods may demand substantively new algorithms or modifications. Miller (1992), for example, offers an incremental algorithm for linear and generalized linear modeling that is suitable for the analysis of massive data. Similarly, Baglama and Reichel (2005) present a new algorithm for a truncated singular value decomposition which may provide a sufficient alternative to a complete principal component analysis in many applications. Our framework provides an effective platform for further development and parallelization of statistical computing methodology.

Efron (2005) notes, “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.” We believe that “classical statistics” should include “computational statistics.” Researchers were able to engage the Netflix data challenge on modern hardware and make advances in predicting ratings. However, the fact that many statisticians struggle to engage massive data research opportunities shows that new computational tools are essential. One contribution of our work is to level the playing field, making it more likely that statisticians with diverse backgrounds and resources can all participate.

Ihaka and Temple Lang (2008) argue that although R has had a tremendous impact on statistics, we should not assume that significant future development of the language is a given. Major changes risk disrupting the user community, slowing the evolution of the environment. At the same time, the nature of applied research in the field of statistics continues to increase demand for advanced statistical computing capabilities, including the analysis of massive data and flexible parallel programming. The R packages introduced in this paper offer (a) short-term solutions to current size limitations of the R environment and (b) a simple, elegant framework for portable parallel computing. However, these solutions also point towards a general design for scalable statistical computing that could be used in future statistical computing environments or a future major revision of R itself.

References

Adler D, Gläser C, Nenadic O, Oehlschlägel J, Zucchini W (2013). ***ff**: Memory-Efficient Storage of Large Data on Disk and Fast Access Functions.* R package version 2.2-12, URL `http://CRAN.R-project.org/package=ff`.

Baglama J, Reichel L (2005). “Augmented Implicitly Restarted Lanczos Bidiagonalization Methods.” *SIAM Journal of Scientific Computing*, **27**(1), 19–42.

Baglama J, Reichel L (2012). ***irlba**: Fast Partial SVD by Implicitly-Restarted Lanczos Bidiagonalization.* R package version 1.0.2, URL http://CRAN.R-project.org/package=irlba.

Bennet J, Lanning S (2007). "The Netflix Prize." In *Proceedings of the KDD Cup and Workshop, San Jose, California.* URL http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf.

Boral H, Alexander W, Clay L, Copeland G, Danforth S, Franklin M, Hart B, Smith M, Valduriez P (1990). "Prototyping Bubba, A Highly Parallel Database System." *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), 4–24.

Chen Q, Therber A, Hsu M, Zeller H, Zhang B, Wu R (2009). "Efficiently Support MapReduce-like Computation Models Inside Parallel DBMS." In *Proceedings of the 2009 International Database Engineering and Applications Symposium*, pp. 43–53. ACM, New York.

Dawes B, *et al.* (2013). "The **Boost** C++ Libraries." URL http://www.boost.org/.

DeWitt DJ, Gerber RH, Graefe G, Heytens ML, Kumar KB, Muralikrishna M (1986). "GAMMA – A High Performance Dataflow Database Machine." In *Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB'86), August 25–28, Kyoto, Japan*, pp. 228–237. URL http://www.vldb.org/conf/1986/P228.PDF.

Efron B (2005). "Bayesians, Frequentists, and Scientists." *Journal of the American Stistical Association*, **100**(469), 1–5.

Emerson JW, Kane MJ (2013a). ***biganalytics**: A Library of Utilities for* `big.matrix` *Objects of Package* ***bigmemory**.* R package version 1.1.1, URL http://CRAN.R-project.org/package=biganalytics.

Emerson JW, Kane MJ (2013b). "The **bigmemory** Project Website." URL http://www.bigmemory.org/.

Emerson JW, Kane MJ, Eddelbuettel D, Allaire JJ, Francois R (2013). ***BH**: The Boost C++ Libraries.* R package version 1.51.0-3, URL http://CRAN.R-project.org/package=BH.

Gaujoux R (2013). ***NMF**: Algorithms and Framework for Nonnegative Matrix Factorization (NMF).* R package version 0.17, URL http://CRAN.R-project.org/package=NMF.

Gaujoux R, Seoighe C (2010). "A Flexible R Package for Nonnegative Matrix Factorization." *BMC Bioinformatics*, **11**(1), 367.

Ihaka R, Temple Lang D (2008). "Back to the Future: Lisp as a Base for a Statistical Computing System." In P Brito (ed.), *COMPSTAT 2008 – Proceedings in Computational Statistics*, pp. 21–33. Physica-Verlag, Heidelberg.

Kane MJ (2013). ***synchronicity**: Boost Mutex Functionality for R.* R package version 1.1.2, URL http://CRAN.R-project.org/package=synchronicity.

Kane MJ, Emerson JW (2013a). The Preprocessed Airline On-Time Performance Data, URL http://euler.stat.yale.edu/~mjk56/Airline.tar.bz2.

Kane MJ, Emerson JW (2013b). ***bigtabulate**: `table`-, `tapply`-, and `split`-like Functionality for `matrix` and `big.matrix` Objects.* R package version 1.1.2, URL http://CRAN.R-project.org/package=bigtabulate.

Kane MJ, Emerson JW, Haverty P (2013a). ***bigmemory**: Manage Massive Matrices with Shared Memory and Memory-Mapped Files.* R package version 4.4.5, URL http://CRAN.R-project.org/package=bigmemory.

Kane MJ, Lewis BW, Emerson JW (2013b). ***bigalgebra**: **BLAS** and **LAPACK** Routines for Native R Matrices and `big.matrix` objects.* R package version 0.8.2, URL https://R-Forge.R-project.org/R/?group_id=556.

Kath R (1993). Managing Memory-Mapped Files, URL http://msdn.microsoft.com/en-us/library/ms810613.aspx.

Lewis BW (2012). ***doRedis**: Foreach Parallel Adapter for the **rredis** Package.* R package version 1.0.5, URL https://github.com/bwlewis/doRedis.

Lumley T (2013). ***biglm**: Bounded Memory Linear and Generalized Linear Models.* R package version 0.9-1, URL http://CRAN.R-project.org/package=biglm.

Miller AJ (1992). "Algorithm AS 274: Least Squares Routines to Supplement Those of Gentlemen." *Journal of the Royal Statistical Society C*, **41**(2), 458–478.

R Core Team (2013a). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

R Core Team (2013b). "R Installation and Administration." URL http://CRAN.R-project.org/doc/manuals/R-admin.html.

Revolution Analytics (2013a). ***doMC**: Foreach Parallel Adaptor for the **multicore** Package.* R package version 1.3.0, URL http://CRAN.R-project.org/package=doMC.

Revolution Analytics (2013b). ***doSNOW**: Foreach Parallel Adaptor for the **snow** Package.* R package version 1.0.7, URL http://CRAN.R-project.org/package=doSNOW.

Revolution Analytics, Weston S (2013a). ***doParallel**: Foreach Parallel Adaptor for the **parallel** Package.* R package version 1.0.3, URL http://CRAN.R-project.org/package=doParallel.

Revolution Analytics, Weston S (2013b). ***foreach**: Foreach Looping Construct for R.* R package version 1.4.1, URL http://CRAN.R-project.org/package=foreach.

RITA (2009). "The Airline On-Time Performance Data Set Website." Research and Innovation Technology Administration, Bureau of Transportation Statistics, URL http://stat-computing.org/dataexpo/2009/.

SAS Institute Inc (2011). *The SAS System, Version 9.3.* SAS Institute Inc., Cary, NC. URL http://www.sas.com/.

SQLite Development Team (2013). "**SQLite** Database Engine." URL http://www.sqlite.org/.

The Apache Software Foundation (2013). "Apache **Hadoop**." URL `http://hadoop.apache.org/`.

The MathWorks, Inc (2011). *MATLAB – The Language of Technical Computing, Version R2011b*. The MathWorks, Inc., Natick, Massachusetts. URL `http://www.mathworks.com/products/matlab/`.

Tierney L, Rossini AJ, Li N, Sevcikova H (2013). ***snow**: Simple Network of Workstations*. R package version 0.3-13, URL `http://CRAN.R-project.org/package=snow`.

Urbanek S (2011). ***multicore**: Parallel Processing of R Code on Machines with Multiple Cores or CPUs*. R package version 0.1-7, URL `http://CRAN.R-project.org/package=multicore`.

Weston S (2013). ***doMPI**: Foreach Parallel Adaptor for the **Rmpi** Package*. R package version 0.2, URL `http://CRAN.R-project.org/package=doMPI`.

Wickham H (2011). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**(1), 1–29. URL `http://www.jstatsoft.org/v40/i01/`.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL `http://cran.r-project.org/doc/Rnews/Rnews_2002-2.pdf`.

Yu H (2013). "The **Rmpi** R package." R package version 0.6-3, URL `http://CRAN.R-project.org/package=Rmpi`.

Affiliation:

Michael J. Kane
The Yale Center for Analytical Sciences
Department of Biostatistics
Yale University
300 George Street
Suite 555
New Haven, CT 06511, United States of America
E-mail: `michael.kane@yale.edu`

John W. Emerson
Department of Statistics
Yale University
24 Hillhouse Avenue
New Haven, CT 06511, United States of America
E-mail: `john.emerson@yale.edu`

Stephen Weston
Department of Computer Science
Yale University
51 Prospect Street
New Haven, CT 06511, United States of America
E-mail: `stephen.weston@yale.edu`

Journal of Statistical Software `http://www.jstatsoft.org/`
published by the American Statistical Association `http://www.amstat.org/`

Volume 55, Issue 14 *Submitted:* 2012-09-05
November 2013 *Accepted:* 2013-04-18