Parallel R

Dr. Zahid Ansari

Table of Contents


```
setwd("D:/AMU Computer Science/Courses/Big Data Analytics/Big Data Analytics Using R/Ch3")
```

- In this part, we will introduce you to the concept of parallelism in R.
- We will focus here on explicit methods for parallel computation, in which users are capable of controlling the parallelization on a single machine.
- Our motivation for parallel computing in R comes from the simple fact that many data-processing operations tend to be very similar, and some of them are extremely time-consuming, especially when using for loops on large datasets or when computing models with multiple different parameters.

- It is generally accepted that a function, which runs for a few minutes while computing an embarrassingly parallel problem, should be spread across several cores, if possible, to reduce the processing time.
- Another reason to implement parallelism in your data manipulation tasks in R is that most currently available, commercially-sold PCs are equipped with more than one CPU. R, however, by default, uses only one, so it is worth making the most of the available architecture, and forcing R to explicitly delegate some work to other cores.
- There are a number of good online resources, which elaborate on parallel computing in R.
- The obvious destination is CRAN Task View – High Performance Computing available at https://cran.r-project.org/web/views/HighPerformanceComputing.html.
- The Task View lists all major packages, known for supporting parallelism in R, and provides brief descriptions about their most essential functionalities.

1 From bigmemory to faster computations

- In the previous section we ran through a number of bigmemory functions which helped us make the most of available RAM.

- The library also contains several useful functions optimized for objects of the S4 class and provides fast computations such as colmean() or colrange() and many others.

- In this section, we will compare the performance of these functions with their base R and parallel implementations by using several methods and R packages that support parallelism.

- For testing purposes, we will try to obtain mean values for each column in the larger version (over 4 mln rows) of the National Energy Efficiency Data-Framework (NEED), which we used previously. If you execute the following code on a smaller sample of NEED data, make sure to use the descriptor file need_data.desc instead of the need_big.desc. Alternatively, you may obtain the full dataset as explained earlier and create a large big.matrix object with the name of the descriptor file set to need_big.desc.

- We will begin from the bigmemory functions to set the performance base levels for comparison. As explained previously, you can use the descriptor file to import the data into R quickly and then use the colmean() function from the biganalytics package to calculate mean values for each column:

```
library(bigmemory)
library(biganalytics)
need.big.bm <- attach.resource("need_big.desc")
meanbig.bm1 <- colmean(need.big.bm, na.rm = TRUE)
meanbig.bm1
```

```
##           HH_ID          REGION         IMD_ENG       IMD_WALES
Gcons2005
##    24908.000000        5.608150        2.992479        2.924645
18935.555209
##  Gcons2005Valid       Gcons2006  Gcons2006Valid       Gcons2007
Gcons2007Valid
##        4.628807    18224.250186        4.651149    17663.802314
4.671986
##       Gcons2008  Gcons2008Valid       Gcons2009  Gcons2009Valid
Gcons2010
##    16936.478941        4.677627    15325.408335        4.695835
14933.481125
##  Gcons2010Valid       Gcons2011  Gcons2011Valid       Gcons2012
Gcons2012Valid
##        4.701395    13928.323728        4.712697    13859.192881
4.715668
##       Econs2005  Econs2005Valid       Econs2006  Econs2006Valid
Econs2007
##     4655.125721        3.906594     4507.868495        3.925484
4448.026137
##  Econs2007Valid       Econs2008  Econs2008Valid       Econs2009
Econs2009Valid
##        3.936164     4248.415982        3.931125     4139.857303
3.932771
##       Econs2010  Econs2010Valid       Econs2011  Econs2011Valid
Econs2012
##     4076.476173        3.943912     4013.730804        1.000000
3972.074709
##  Econs2012Valid      E7Flag2012  MAIN_HEAT_FUEL        PROP_AGE
PROP_TYPE
##        1.997952        1.000000        1.145920      103.016903
103.391850
## FLOOR_AREA_BAND         EE_BAND      LOFT_DEPTH       WALL_CONS
CWI
##        2.310288        3.063776       26.293024        1.352765
1.000000
##        CWI_YEAR              LI         LI_YEAR          BOILER
BOILER_YEAR
##     2007.499250        1.000000     2009.015964        1.000000
2008.793523
```

- It took 6.22 seconds to run this function on my computer. In the following examples, we will review other more optimized methods, including selected functions that support parallelism.

2 An apply() example with the big.matrix object

- You can also obtain the same output by invoking the apply() function from the same biganalytics package.

- The apply() function from biganalytics is a generalization of the base R apply() function that additionally supports the S4 class object of big.matrix type.
- Apart from this subtle difference, both functions are identical. As the big.matrix object is a custom data structure, the apply() function deals differently with the memory overhead associated with extracting data from this S4 class, and, for that reason, we may expect that this implementation will actually run slower than colmean():

```
meanbig.bm2 <- apply(need.big.bm, 2, mean, na.rm=TRUE)
meanbig.bm2

##           HH_ID          REGION         IMD_ENG       IMD_WALES
Gcons2005
##    24908.000000        5.608150        2.992479        2.924645
18935.555209
##  Gcons2005Valid       Gcons2006  Gcons2006Valid       Gcons2007
Gcons2007Valid
##        4.628807    18224.250186        4.651149    17663.802314
4.671986
##       Gcons2008  Gcons2008Valid       Gcons2009  Gcons2009Valid
Gcons2010
##    16936.478941        4.677627    15325.408335        4.695835
14933.481125
##  Gcons2010Valid       Gcons2011  Gcons2011Valid       Gcons2012
Gcons2012Valid
##        4.701395    13928.323728        4.712697    13859.192881
4.715668
##       Econs2005  Econs2005Valid       Econs2006  Econs2006Valid
Econs2007
##     4655.125721        3.906594     4507.868495        3.925484
4448.026137
##  Econs2007Valid       Econs2008  Econs2008Valid       Econs2009
Econs2009Valid
##        3.936164     4248.415982        3.931125     4139.857303
3.932771
##       Econs2010  Econs2010Valid       Econs2011  Econs2011Valid
Econs2012
##     4076.476173        3.943912     4013.730804        1.000000
3972.074709
##  Econs2012Valid      E7Flag2012  MAIN_HEAT_FUEL        PROP_AGE
PROP_TYPE
##        1.997952        1.000000        1.145920      103.016903
103.391850
## FLOOR_AREA_BAND         EE_BAND      LOFT_DEPTH       WALL_CONS
CWI
##        2.310288        3.063776       26.293024        1.352765
1.000000
##        CWI_YEAR              LI         LI_YEAR          BOILER
BOILER_YEAR
```

```
##     2007.499250        1.000000     2009.015964        1.000000
2008.793523
```

- The preceding process completed in 8.21 seconds, so almost 2 seconds slower than with the colmean() method.

2.1 The apply() family of functions

- In general, apply() methods can save quite a lot of your precious data processing time, and you don't need to write loops to calculate the same statistics over all columns, rows, or other dimensions of your data structures.

2.2 A for() loop example with the ffdf object

- Going back to our mean estimations for each column and performance comparison between available methods, let's see whether there is any improvement in processing speed if we wanted to apply a for() loop on the 4-million-row NEED file imported through the ff package which we described earlier.
- Assuming that the relevant ffdf object has been already created in the R environment, we may now run the following code:

```
library(ff)
library(ffbase)
need.big.ff <- read.table.ffdf(file="need_data.csv", sep=",", VERBOSE=TRUE,
header=TRUE, next.rows=100000, colClasses=NA)

## read.table.ffdf 1..49815 (49815)  csv-read=1.76sec ffdf-write=0.91sec
##  csv-read=1.76sec  ffdf-write=0.91sec  TOTAL=2.67sec

meanbig.ff <- list()
for(i in 1:ncol(need.big.ff)) {
    meanbig.ff[[i]] <- mean.ff(need.big.ff[[i]], na.rm=TRUE)
}
meanbig.ff

## [[1]]
## [1] 24908
##
## [[2]]
## [1] 5.60815
##
## [[3]]
## [1] 2.992479
##
## [[4]]
## [1] 2.924645
##
## [[5]]
## [1] 18935.56
##
## [[6]]
## [1] 4.628807
```

```
##
## [[7]]
## [1] 18224.25
##
## [[8]]
## [1] 4.651149
##
## [[9]]
## [1] 17663.8
##
## [[10]]
## [1] 4.671986
##
## [[11]]
## [1] 16936.48
##
## [[12]]
## [1] 4.677627
##
## [[13]]
## [1] 15325.41
##
## [[14]]
## [1] 4.695835
##
## [[15]]
## [1] 14933.48
##
## [[16]]
## [1] 4.701395
##
## [[17]]
## [1] 13928.32
##
## [[18]]
## [1] 4.712697
##
## [[19]]
## [1] 13859.19
##
## [[20]]
## [1] 4.715668
##
## [[21]]
## [1] 4655.126
##
## [[22]]
## [1] 3.906594
##
## [[23]]
```

```
## [1] 4507.868
##
## [[24]]
## [1] 3.925484
##
## [[25]]
## [1] 4448.026
##
## [[26]]
## [1] 3.936164
##
## [[27]]
## [1] 4248.416
##
## [[28]]
## [1] 3.931125
##
## [[29]]
## [1] 4139.857
##
## [[30]]
## [1] 3.932771
##
## [[31]]
## [1] 4076.476
##
## [[32]]
## [1] 3.943912
##
## [[33]]
## [1] 4013.731
##
## [[34]]
## [1] 1
##
## [[35]]
## [1] 3972.075
##
## [[36]]
## [1] 1.997952
##
## [[37]]
## [1] 1
##
## [[38]]
## [1] 1.14592
##
## [[39]]
## [1] 103.0169
##
```

```
## [[40]]
## [1] 103.3918
##
## [[41]]
## [1] 2.310288
##
## [[42]]
## [1] 3.063776
##
## [[43]]
## [1] 26.29302
##
## [[44]]
## [1] 1.352765
##
## [[45]]
## [1] 1
##
## [[46]]
## [1] 2007.499
##
## [[47]]
## [1] 1
##
## [[48]]
## [1] 2009.016
##
## [[49]]
## [1] 1
##
## [[50]]
## [1] 2008.794
```

- In the preceding snippet, we've used the mean.ff() function, which is simply a the S3 method for the class ff that derives from the generic mean() function in the base R.
- The for() loop completed in 7.72 seconds, providing only a slight improvement when compared to the performance of apply() used on the big.matrix object, but it was still slower than colmean() through the biganalytics package. How can we then boost the performance in a more significant way?

2.3 Using apply() and for() loop examples on a data.frame

One main reason why we have been using the bigmemory and ff/ffdf packages extensively in this chapter is their ability to process out-of-memory data directly from the R console.

But by mapping their custom data structures to raw data stored on disk, we consciously compromise the performance of our operations.

For datasets that can fit within the RAM boundaries, if you have the comfort of working on a large server, or if you are using some of the cloud computing services (and you will when you get to Online Chapter, Pushing R Further https://www.packtpub.com/sites/default/files/downloads/5396457OSPushingRFurther.pdf), you may also import the data directly to the physical memory, and use either one of the generic functions such as colMeans() or for() loops on a created data frame object:

```
need.big.df <- read.csv("need_data.csv")
x1 = list()
for(i in 1:ncol(need.big.df)) {
  x1[i] <- mean(need.big.df[,i], na.rm = TRUE)
}
x1
```

```
## [[1]]
## [1] 24908
##
## [[2]]
## [1] 5.60815
##
## [[3]]
## [1] 2.992479
##
## [[4]]
## [1] 2.924645
##
## [[5]]
## [1] 18935.56
##
## [[6]]
## [1] 4.628807
##
## [[7]]
## [1] 18224.25
##
## [[8]]
## [1] 4.651149
##
## [[9]]
## [1] 17663.8
##
## [[10]]
## [1] 4.671986
##
## [[11]]
## [1] 16936.48
##
## [[12]]
## [1] 4.677627
##
```

```
## [[13]]
## [1] 15325.41
##
## [[14]]
## [1] 4.695835
##
## [[15]]
## [1] 14933.48
##
## [[16]]
## [1] 4.701395
##
## [[17]]
## [1] 13928.32
##
## [[18]]
## [1] 4.712697
##
## [[19]]
## [1] 13859.19
##
## [[20]]
## [1] 4.715668
##
## [[21]]
## [1] 4655.126
##
## [[22]]
## [1] 3.906594
##
## [[23]]
## [1] 4507.868
##
## [[24]]
## [1] 3.925484
##
## [[25]]
## [1] 4448.026
##
## [[26]]
## [1] 3.936164
##
## [[27]]
## [1] 4248.416
##
## [[28]]
## [1] 3.931125
##
## [[29]]
## [1] 4139.857
```

```
##
## [[30]]
## [1] 3.932771
##
## [[31]]
## [1] 4076.476
##
## [[32]]
## [1] 3.943912
##
## [[33]]
## [1] 4013.731
##
## [[34]]
## [1] 1
##
## [[35]]
## [1] 3972.075
##
## [[36]]
## [1] 1.997952
##
## [[37]]
## [1] 1
##
## [[38]]
## [1] 1.14592
##
## [[39]]
## [1] 103.0169
##
## [[40]]
## [1] 103.3918
##
## [[41]]
## [1] 2.310288
##
## [[42]]
## [1] 3.063776
##
## [[43]]
## [1] 26.29302
##
## [[44]]
## [1] 1.352765
##
## [[45]]
## [1] 1
##
## [[46]]
```

```
## [1] 2007.499
##
## [[47]]
## [1] 1
##
## [[48]]
## [1] 2009.016
##
## [[49]]
## [1] 1
##
## [[50]]
## [1] 2008.794
```

The for() loop with the base mean() function with its completion time of 5.24 seconds, was faster than any other method presented previously. The catch is that there was quite a substantial peak of memory usage and we are now holding a large object in the workspace. But colMeans() approach from base R beats all others with only 2.9 seconds:

```
x2 <- colMeans(need.big.df, na.rm = TRUE)
x2

##          HH_ID          REGION         IMD_ENG       IMD_WALES
Gcons2005
##   24908.000000        5.608150        2.992479        2.924645
18935.555209
##  Gcons2005Valid       Gcons2006  Gcons2006Valid       Gcons2007
Gcons2007Valid
##       4.628807    18224.250186        4.651149    17663.802314
4.671986
##      Gcons2008  Gcons2008Valid       Gcons2009  Gcons2009Valid
Gcons2010
##   16936.478941        4.677627    15325.408335        4.695835
14933.481125
##  Gcons2010Valid       Gcons2011  Gcons2011Valid       Gcons2012
Gcons2012Valid
##       4.701395    13928.323728        4.712697    13859.192881
4.715668
##      Econs2005  Econs2005Valid       Econs2006  Econs2006Valid
Econs2007
##    4655.125721        3.906594     4507.868495        3.925484
4448.026137
##  Econs2007Valid       Econs2008  Econs2008Valid       Econs2009
Econs2009Valid
##       3.936164     4248.415982        3.931125     4139.857303
3.932771
##      Econs2010  Econs2010Valid       Econs2011  Econs2011Valid
Econs2012
##    4076.476173        3.943912     4013.730804        1.000000
3972.074709
```

```
##   Econs2012Valid       E7Flag2012  MAIN_HEAT_FUEL         PROP_AGE
PROP_TYPE
##         1.997952         1.000000        1.145920       103.016903
103.391850
## FLOOR_AREA_BAND          EE_BAND      LOFT_DEPTH        WALL_CONS
CWI
##         2.310288         3.063776       26.293024         1.352765
1.000000
##         CWI_YEAR               LI         LI_YEAR           BOILER
BOILER_YEAR
##      2007.499250         1.000000     2009.015964         1.000000
2008.793523
```

However, as the colMeans() function is in fact equivalent to the apply() method, let's test to see if the simplified version of apply() in the form of sapply() can keep up the pace:

```
x3 <- sapply(need.big.df, mean, na.rm = TRUE)
x3

##            HH_ID           REGION         IMD_ENG        IMD_WALES
Gcons2005
##     24908.000000         5.608150        2.992479         2.924645
18935.555209
##   Gcons2005Valid        Gcons2006  Gcons2006Valid        Gcons2007
Gcons2007Valid
##         4.628807     18224.250186        4.651149     17663.802314
4.671986
##        Gcons2008   Gcons2008Valid       Gcons2009   Gcons2009Valid
Gcons2010
##     16936.478941         4.677627    15325.408335         4.695835
14933.481125
##   Gcons2010Valid        Gcons2011  Gcons2011Valid        Gcons2012
Gcons2012Valid
##         4.701395     13928.323728        4.712697     13859.192881
4.715668
##        Econs2005   Econs2005Valid       Econs2006   Econs2006Valid
Econs2007
##      4655.125721         3.906594     4507.868495         3.925484
4448.026137
##   Econs2007Valid        Econs2008  Econs2008Valid        Econs2009
Econs2009Valid
##         3.936164      4248.415982        3.931125      4139.857303
3.932771
##        Econs2010   Econs2010Valid       Econs2011   Econs2011Valid
Econs2012
##      4076.476173         3.943912     4013.730804         1.000000
3972.074709
##   Econs2012Valid       E7Flag2012  MAIN_HEAT_FUEL         PROP_AGE
PROP_TYPE
##         1.997952         1.000000        1.145920       103.016903
```

```
103.391850
## FLOOR_AREA_BAND         EE_BAND      LOFT_DEPTH       WALL_CONS
CWI
##        2.310288        3.063776       26.293024        1.352765
1.000000
##        CWI_YEAR              LI         LI_YEAR          BOILER
BOILER_YEAR
##     2007.499250        1.000000     2009.015964        1.000000
2008.793523
```

It took almost 5.2 seconds for sapply() to calculate the means for all columns in our data, which is much slower than through colMeans(). We may try to optimize the speed of the apply() function by parallelizing its execution explicitly through the parallel package.

2.4 A parallel package example

The parallel package has come as an integral part of the core R installation since the R 2.14.0 version, and it has been built on two other popular R packages that support parallel data processing: multicore (authored by Simon Urbanek) and snow (Simple Network of Workstations, created by Tierney, Rossini, Li, and Sevcikova). In fact, multicore has already been discontinued and removed from the CRAN repository, as the parallel package took over all its essential components. The snow package is still available on CRAN, and it may be useful in certain, but limited, circumstances.

The parallel library, however, extends their functionalities by allowing greater support for random-number generation. The package can be suitable for parallelizing repetitive jobs on unrelated chunks of data, and computations that do not need to communicate with, and between, one another

The computational model adopted by parallel is similar to approaches known from earlier packages for snow, and it's based on the relationship between master and worker processes. The details of this model can be found in a short R manual dedicated to the parallel package and is available at http://stat.ethz.ch/R-manual/Rdevel/library/parallel/doc/parallel.pdf.

In order to perform any parallel processing jobs, it might first be advisable to know how many physical CPUs (or cores) are available on the machine that runs R. This can be achieved in the parallel package with the detectCores() function:

```
library(parallel)
detectCores()

## [1] 8
```

It is important here to be aware that the returned number of cores may not be equal to the actual number of available logical cores (for example in Windows), or CPU accessible to a specific user on restricted multi-user systems. The parallel package, by default, facilitates clusters communicating over two types of sockets: SOCK and FORK.

The SOCK cluster, operationalized through the makePSOCKcluster() function, is simply an enhanced implementation of the makeSOCKcluster() command known from the snow package.

The FORK cluster, on the other hand, originates from the multicore package and allows the creation of multiple R processes by copying the master process completely including R GUI elements such as an R console and devices. The forking is generally available on most non-Windows R distributions. Other types of clusters can be created using the snow package (for example through MPI or NWS connections) or with makeCluster() in the parallel package, which will call snow, provided it's included in the search path.

In parallel, you may use either makePSOCKcluster() or makeCluster() functions to create a SOCK cluster:

```
library(snow)

##
## Attaching package: 'snow'

## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, clusterSplit, makeCluster, parApply,
##     parCapply, parLapply, parRapply, parSapply, splitIndices,
##     stopCluster

cl <- makeCluster(7, type = "SOCK")
cl

## socket cluster with 7 nodes on host 'localhost'
```

Generally, it is advisable to create clusters with n number of nodes, where n = detectCores() - 1, hence seven nodes in our example. This approach allows us to benefit from multi-threading, without putting an excessive pressure on other processes ornapplications that may be run in parallel.

The parallel package allows the execution of apply() operations on each node in the cluster through the clusterApply() function and parallelized implementations of the apply() family of functions known from the multicore and snow packages: parLapply(), parSapply(), and parApply().

```
meanbig <- clusterApply(cl, need.big.df, fun=mean, na.rm=TRUE)
meanbig

## [[1]]
## [1] 24908
##
## [[2]]
## [1] 5.60815
##
## [[3]]
```

```
## [1] 2.992479
##
## [[4]]
## [1] 2.924645
##
## [[5]]
## [1] 18935.56
##
## [[6]]
## [1] 4.628807
##
## [[7]]
## [1] 18224.25
##
## [[8]]
## [1] 4.651149
##
## [[9]]
## [1] 17663.8
##
## [[10]]
## [1] 4.671986
##
## [[11]]
## [1] 16936.48
##
## [[12]]
## [1] 4.677627
##
## [[13]]
## [1] 15325.41
##
## [[14]]
## [1] 4.695835
##
## [[15]]
## [1] 14933.48
##
## [[16]]
## [1] 4.701395
##
## [[17]]
## [1] 13928.32
##
## [[18]]
## [1] 4.712697
##
## [[19]]
## [1] 13859.19
##
```

```
## [[20]]
## [1] 4.715668
##
## [[21]]
## [1] 4655.126
##
## [[22]]
## [1] 3.906594
##
## [[23]]
## [1] 4507.868
##
## [[24]]
## [1] 3.925484
##
## [[25]]
## [1] 4448.026
##
## [[26]]
## [1] 3.936164
##
## [[27]]
## [1] 4248.416
##
## [[28]]
## [1] 3.931125
##
## [[29]]
## [1] 4139.857
##
## [[30]]
## [1] 3.932771
##
## [[31]]
## [1] 4076.476
##
## [[32]]
## [1] 3.943912
##
## [[33]]
## [1] 4013.731
##
## [[34]]
## [1] 1
##
## [[35]]
## [1] 3972.075
##
## [[36]]
## [1] 1.997952
```

```
##
## [[37]]
## [1] 1
##
## [[38]]
## [1] 1.14592
##
## [[39]]
## [1] 103.0169
##
## [[40]]
## [1] 103.3918
##
## [[41]]
## [1] 2.310288
##
## [[42]]
## [1] 3.063776
##
## [[43]]
## [1] 26.29302
##
## [[44]]
## [1] 1.352765
##
## [[45]]
## [1] 1
##
## [[46]]
## [1] 2007.499
##
## [[47]]
## [1] 1
##
## [[48]]
## [1] 2009.016
##
## [[49]]
## [1] 1
##
## [[50]]
## [1] 2008.794
```

Unfortunately, the clusterApply() approach is quite slow and completes the job in 13.74 seconds. The parSapply() implementation returning the output is much faster and returns the output in 6.71 seconds:

```
meanbig2 <- parSapply(cl, need.big.df, FUN = mean, na.rm=TRUE)
meanbig2
```

```
##           HH_ID          REGION         IMD_ENG       IMD_WALES 
Gcons2005 
##    24908.000000        5.608150        2.992479        2.924645 
18935.555209 
##  Gcons2005Valid       Gcons2006  Gcons2006Valid       Gcons2007 
Gcons2007Valid 
##        4.628807    18224.250186        4.651149    17663.802314 
4.671986 
##       Gcons2008  Gcons2008Valid       Gcons2009  Gcons2009Valid 
Gcons2010 
##    16936.478941        4.677627    15325.408335        4.695835 
14933.481125 
##  Gcons2010Valid       Gcons2011  Gcons2011Valid       Gcons2012 
Gcons2012Valid 
##        4.701395    13928.323728        4.712697    13859.192881 
4.715668 
##       Econs2005  Econs2005Valid       Econs2006  Econs2006Valid 
Econs2007 
##     4655.125721        3.906594     4507.868495        3.925484 
4448.026137 
##  Econs2007Valid       Econs2008  Econs2008Valid       Econs2009 
Econs2009Valid 
##        3.936164     4248.415982        3.931125     4139.857303 
3.932771 
##       Econs2010  Econs2010Valid       Econs2011  Econs2011Valid 
Econs2012 
##     4076.476173        3.943912     4013.730804        1.000000 
3972.074709 
##  Econs2012Valid      E7Flag2012  MAIN_HEAT_FUEL        PROP_AGE 
PROP_TYPE 
##        1.997952        1.000000        1.145920      103.016903 
103.391850 
## FLOOR_AREA_BAND         EE_BAND      LOFT_DEPTH       WALL_CONS 
CWI 
##        2.310288        3.063776       26.293024        1.352765 
1.000000 
##        CWI_YEAR              LI         LI_YEAR          BOILER 
BOILER_YEAR 
##     2007.499250        1.000000     2009.015964        1.000000 
2008.793523
```

In addition to the apply() operations presented earlier, the parallel package contains the mclapply() function, which is a parallelized version of lapply() relying on forking (not available on Windows unless mc.cores = 1).

In the following example we will compare the performance of mclapply() with differing number of cores (the mc.cores argument from 1 to 4):

```
meanbig3 <- mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 1)
meanbig3
```

```
## $HH_ID
## [1] 24908
##
## $REGION
## [1] 5.60815
##
## $IMD_ENG
## [1] 2.992479
##
## $IMD_WALES
## [1] 2.924645
##
## $Gcons2005
## [1] 18935.56
##
## $Gcons2005Valid
## [1] 4.628807
##
## $Gcons2006
## [1] 18224.25
##
## $Gcons2006Valid
## [1] 4.651149
##
## $Gcons2007
## [1] 17663.8
##
## $Gcons2007Valid
## [1] 4.671986
##
## $Gcons2008
## [1] 16936.48
##
## $Gcons2008Valid
## [1] 4.677627
##
## $Gcons2009
## [1] 15325.41
##
## $Gcons2009Valid
## [1] 4.695835
##
## $Gcons2010
## [1] 14933.48
##
## $Gcons2010Valid
## [1] 4.701395
##
## $Gcons2011
## [1] 13928.32
```

```
##
## $Gcons2011Valid
## [1] 4.712697
##
## $Gcons2012
## [1] 13859.19
##
## $Gcons2012Valid
## [1] 4.715668
##
## $Econs2005
## [1] 4655.126
##
## $Econs2005Valid
## [1] 3.906594
##
## $Econs2006
## [1] 4507.868
##
## $Econs2006Valid
## [1] 3.925484
##
## $Econs2007
## [1] 4448.026
##
## $Econs2007Valid
## [1] 3.936164
##
## $Econs2008
## [1] 4248.416
##
## $Econs2008Valid
## [1] 3.931125
##
## $Econs2009
## [1] 4139.857
##
## $Econs2009Valid
## [1] 3.932771
##
## $Econs2010
## [1] 4076.476
##
## $Econs2010Valid
## [1] 3.943912
##
## $Econs2011
## [1] 4013.731
##
## $Econs2011Valid
```

```
## [1] 1
##
## $Econs2012
## [1] 3972.075
##
## $Econs2012Valid
## [1] 1.997952
##
## $E7Flag2012
## [1] 1
##
## $MAIN_HEAT_FUEL
## [1] 1.14592
##
## $PROP_AGE
## [1] 103.0169
##
## $PROP_TYPE
## [1] 103.3918
##
## $FLOOR_AREA_BAND
## [1] 2.310288
##
## $EE_BAND
## [1] 3.063776
##
## $LOFT_DEPTH
## [1] 26.29302
##
## $WALL_CONS
## [1] 1.352765
##
## $CWI
## [1] 1
##
## $CWI_YEAR
## [1] 2007.499
##
## $LI
## [1] 1
##
## $LI_YEAR
## [1] 2009.016
##
## $BOILER
## [1] 1
##
## $BOILER_YEAR
## [1] 2008.794
```

```r
library(tictoc)
tic("1 Core")
mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 1)
```

```
## $HH_ID
## [1] 24908
##
## $REGION
## [1] 5.60815
##
## $IMD_ENG
## [1] 2.992479
##
## $IMD_WALES
## [1] 2.924645
##
## $Gcons2005
## [1] 18935.56
##
## $Gcons2005Valid
## [1] 4.628807
##
## $Gcons2006
## [1] 18224.25
##
## $Gcons2006Valid
## [1] 4.651149
##
## $Gcons2007
## [1] 17663.8
##
## $Gcons2007Valid
## [1] 4.671986
##
## $Gcons2008
## [1] 16936.48
##
## $Gcons2008Valid
## [1] 4.677627
##
## $Gcons2009
## [1] 15325.41
##
## $Gcons2009Valid
## [1] 4.695835
##
## $Gcons2010
## [1] 14933.48
##
## $Gcons2010Valid
```

```
## [1] 4.701395
##
## $Gcons2011
## [1] 13928.32
##
## $Gcons2011Valid
## [1] 4.712697
##
## $Gcons2012
## [1] 13859.19
##
## $Gcons2012Valid
## [1] 4.715668
##
## $Econs2005
## [1] 4655.126
##
## $Econs2005Valid
## [1] 3.906594
##
## $Econs2006
## [1] 4507.868
##
## $Econs2006Valid
## [1] 3.925484
##
## $Econs2007
## [1] 4448.026
##
## $Econs2007Valid
## [1] 3.936164
##
## $Econs2008
## [1] 4248.416
##
## $Econs2008Valid
## [1] 3.931125
##
## $Econs2009
## [1] 4139.857
##
## $Econs2009Valid
## [1] 3.932771
##
## $Econs2010
## [1] 4076.476
##
## $Econs2010Valid
## [1] 3.943912
##
```

```
## $Econs2011
## [1] 4013.731
##
## $Econs2011Valid
## [1] 1
##
## $Econs2012
## [1] 3972.075
##
## $Econs2012Valid
## [1] 1.997952
##
## $E7Flag2012
## [1] 1
##
## $MAIN_HEAT_FUEL
## [1] 1.14592
##
## $PROP_AGE
## [1] 103.0169
##
## $PROP_TYPE
## [1] 103.3918
##
## $FLOOR_AREA_BAND
## [1] 2.310288
##
## $EE_BAND
## [1] 3.063776
##
## $LOFT_DEPTH
## [1] 26.29302
##
## $WALL_CONS
## [1] 1.352765
##
## $CWI
## [1] 1
##
## $CWI_YEAR
## [1] 2007.499
##
## $LI
## [1] 1
##
## $LI_YEAR
## [1] 2009.016
##
## $BOILER
## [1] 1
```

```
##
## $BOILER_YEAR
## [1] 2008.794

toc()

## 1 Core: 0.04 sec elapsed

# tic("2 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 2)
# toc()
# tic("3 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 3)
# toc()
# tic("4 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 4)
# toc()
# tic("5 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 5)
# toc()
# tic("6 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 6)
# toc()
# tic("7 Cores")
# mclapply(need.big.df, FUN = mean, na.rm = TRUE, mc.cores = 7)
# toc()
```

The table below presents average timings of evaluation of the same mclapply() expression with a differing number of mc.cores (from 1 to 4): mc.cores time (in seconds) 1 4.71 2 4.14 3 3.51 4 3.10 It is clear that the increase in cores correlates with the better performance. Note, however, that because the parallel implementation of mclapply() initializes several processes which share the same GUI, it is advisable not to run it in the R GUI or embedded environments, otherwise your machine (and R sessions) may become unresponsive, cause chaos, or even crash. For larger datasets, several parallel R sessions may rapidly increase the memory usage and its pressure, so please be extremely careful when implementing the parallelized apply() family of functions into your data processing workflows. Once the parallel jobs are complete it is a good habit to close all connections with the following statement:

```
stopCluster(cl)
```

The previously mentioned R manual on parallel is available from http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf and presents two very good frequent applications of the package: in bootstrapping and maximum-likelihood estimations. Please feel free to visit the manual and run through the given examples.

2.5 A foreach package example

The foreach() package, authored by Revolution Analytics, Rich Calaway, and Steve Weston, offers an alternative method of implementing for() loops, but without the need to use the loop counter explicitly. It also supports the parallel execution of loops through the doParallel backend and the parallel package. Sticking to our example, with mean estimates for each column of the data, we may apply foreach() in the following manner:

```
library(iterators)
library(foreach)
library(parallel)
library(doParallel)

cl <- makeCluster(7, type = "SOCK")
registerDoParallel(cl)
x4 <- foreach(i = 1:ncol(need.big.df)) %dopar% mean(need.big.df[,i],
na.rm=TRUE)
x4

## [[1]]
## [1] 24908
##
## [[2]]
## [1] 5.60815
##
## [[3]]
## [1] 2.992479
##
## [[4]]
## [1] 2.924645
##
## [[5]]
## [1] 18935.56
##
## [[6]]
## [1] 4.628807
##
## [[7]]
## [1] 18224.25
##
## [[8]]
## [1] 4.651149
##
## [[9]]
## [1] 17663.8
##
## [[10]]
## [1] 4.671986
##
## [[11]]
## [1] 16936.48
##
## [[12]]
## [1] 4.677627
```

```
##
## [[13]]
## [1] 15325.41
##
## [[14]]
## [1] 4.695835
##
## [[15]]
## [1] 14933.48
##
## [[16]]
## [1] 4.701395
##
## [[17]]
## [1] 13928.32
##
## [[18]]
## [1] 4.712697
##
## [[19]]
## [1] 13859.19
##
## [[20]]
## [1] 4.715668
##
## [[21]]
## [1] 4655.126
##
## [[22]]
## [1] 3.906594
##
## [[23]]
## [1] 4507.868
##
## [[24]]
## [1] 3.925484
##
## [[25]]
## [1] 4448.026
##
## [[26]]
## [1] 3.936164
##
## [[27]]
## [1] 4248.416
##
## [[28]]
## [1] 3.931125
##
## [[29]]
```

```
## [1] 4139.857
##
## [[30]]
## [1] 3.932771
##
## [[31]]
## [1] 4076.476
##
## [[32]]
## [1] 3.943912
##
## [[33]]
## [1] 4013.731
##
## [[34]]
## [1] 1
##
## [[35]]
## [1] 3972.075
##
## [[36]]
## [1] 1.997952
##
## [[37]]
## [1] 1
##
## [[38]]
## [1] 1.14592
##
## [[39]]
## [1] 103.0169
##
## [[40]]
## [1] 103.3918
##
## [[41]]
## [1] 2.310288
##
## [[42]]
## [1] 3.063776
##
## [[43]]
## [1] 26.29302
##
## [[44]]
## [1] 1.352765
##
## [[45]]
## [1] 1
##
```

```
## [[46]]
## [1] 2007.499
##
## [[47]]
## [1] 1
##
## [[48]]
## [1] 2009.016
##
## [[49]]
## [1] 1
##
## [[50]]
## [1] 2008.794
```

The job took 5.2 seconds to complete.

In the first part of the listing, we have created a seven node cluster cl, which we registered with the foreach package using registerDoParallel() from the doParallel library-a parallel backend.

You've also probably noticed that the above foreach() statement contains an unfamiliar piece of syntax: %dopar%. It is a binary operator that evaluates an R expression (mean(...)) in parallel in an environment created by the foreach object.

If you wished to run the same call sequentially, you could use the %do% operator instead.

In fact, both implementations return the output of our mean calculations within a very similar time, but the actual timings will obviously depend on the specific computation and available architecture.

The foreach() function contains a number of other useful settings. For example, you can present the output as a vector, matrix, or in any other way, defined by a function set in the .combine argument.

In the following code snippet, we use foreach() with an %do% operator and a .combine argument set to concatenate the values (that is, to present them as a vector):

```
x5 <- foreach(i = 1:ncol(need.big.df), .combine = "c") %do%
mean(need.big.df[,i], na.rm=TRUE)
x5
```

```
##  [1] 24908.000000     5.608150     2.992479     2.924645 18935.555209
##  [6]     4.628807 18224.250186     4.651149 17663.802314     4.671986
## [11] 16936.478941     4.677627 15325.408335     4.695835 14933.481125
## [16]     4.701395 13928.323728     4.712697 13859.192881     4.715668
## [21]  4655.125721     3.906594  4507.868495     3.925484  4448.026137
## [26]     3.936164  4248.415982     3.931125  4139.857303     3.932771
## [31]  4076.476173     3.943912  4013.730804     1.000000  3972.074709
## [36]     1.997952     1.000000     1.145920   103.016903   103.391850
```

```
## [41]     2.310288     3.063776    26.293024     1.352765     1.000000
## [46]  2007.499250     1.000000  2009.015964     1.000000  2008.793523
```

The foreach package is still pretty new to the R community, and it is expected that more functionalities will be added within the next several months.

Steve Weston's guide Using The foreach Package (available from https://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf) contains several simple, and slightly more complex, applications of specific parameters (and their values) which can be set in the foreach() function.

3 The future of parallel processing in R

In the preceding section, we have introduced you to some basics of parallel computing currently available from within R, on a single machine. R is probably not the ideal solution for parallelized operations, but a number of more recent approaches may potentially revolutionize the way R implements parallelism.

3.1 Utilizing Graphics Processing Units with R

Graphics Processing Units (GPUs) are specialized, high-performance electronic circuits that are designed for efficient and fast memory management in computationally demanding tasks, such as image and video rendering, dynamic gameplay, simulations (both 3D or virtual and also statistical), and many others.

Although they are still rarely used in general calculations, a growing number of researchers benefit from GPU acceleration when carrying out repetitive, embarrassingly parallel, computations over multiple parameters. The major disadvantage of GPUs, however, is that they don't generally support Big Data analytics on a single machine owing to their limited access to RAM. Again, it depends what one means by Big Data, and also, their application in the processing of large datasets relies on the architecture in place.

On average, however, parallel computing through GPU can be up to 12 times as fast, compared to parallel jobs using standard CPUs.

The largest companies manufacturing GPU are Intel, NVIDIA, and AMD, and you are probably familiar with some of their products if you ever built a PC yourself, or at least played some computer games.

R also supports parallel computing through GPUs, but obviously you can only make the most of it if your machine is equipped with one of the leading GPUs for example NVIDIA CUDA. If you don't own one, you can quite cheaply create a cloud-computing cluster, for example, on Amazon Elastic Cloud Computing (EC2), which will include graphics processing units.

We will show you how to deploy such an EC2 instance with R and RStudio Server installed in Online Chapter, Pushing R Further (h t t p s : / / w w w . p a c k t p u b . c o m / s i t e s / d e f a u l t / f i l e s / d o w n l o a d s / 5 3 9 6 6 4 5 7 O S P u s h i n g R F u r t h e r . p d f).

As the GPUs need to be programmed, and hence many R users may struggle with their configuration, there are several R packages that facilitate working with CUDA-compatible GPU. One of them is the gputools package authored by Buckner, Seligman, and Wilson. The package requires a recent version of the NVIDIA CUDA toolkit and it contains a set of GPU-optimized statistical methods such as (but not limited to) fitting generalized linear models (the gpuGlm() function), performing hierarchical clustering for vectors (gpuHclust()), computing distances between vectors (gpuDist()), calculating Pearson or Kendall correlation coefficients (gpuCor()), and estimating t-tests (gpuTtest()).

The gputools package can also implement fastICA algorithm (Fast Independent Component Analysis algorithm created by Prof. Aapo Hyvarinen from University of Helsinki, http://www.cs.helsinki.fi/u/ahyvarin/) through CULA Tools (http://www.culatools.com/)-a collection of GPU-supported linear algebra libraries for parallel computing.

More details on the gputools package are available from the following sources: https://cran.r-project.org/web/packages/gputools/index.htmlthe gputools CRAN website with links to manuals and source code https://github.com/nullsatz/gputools/wiki-the gputools GitHub project repo http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/-the gputools website run by Microarray Lab at the Molecular and Behavioral Neuroscience Institute, University of Michigan Apart from gputools, R packages, also provide an interface with OpenCL-a programming language framework used for operating heterogeneous computational platforms based on a variety of CPU, GPU, and other accelerator devices.

The OpenCL package, developed and maintained by Simon Urbanek, allows R users to identify and retrieve a list of OpenCL devices and to execute kernel code that has been compiled for OpenCL directly from the R console.

On the other hand, the gpuR package, created and maintained by Charles Determan Jr., simplifies this task by providing users with ready-made custom gpu and vcl classes which function as wrappers for common R data structures such as vector or matrix.

Without any prior knowledge of OpenCL, R users can easily perform a number of statistical methods through the gpuR package such as estimating row and column sums and arithmetic means, comparing elements of gpuvector and vector objects, calculating covariance and cross-product on gpuMatrix and vclMatrix, distance matrix estimations, eigenvalues computations, and many others.

3.2 CRAN High-Performance Computing Task View

(https://cran.r-project.org/web/views/HighPerformanceComputing.html) lists a few more specialized R packages, which support GPU acceleration. It doesn't, however, mention the rpud package, which was removed from CRAN in late October 2015 owing to maintenance issues. The package, however, has been quite successful in performing several GPU-optimized statistical methods such as hierarchical cluster analysis and classification tasks.

Although it is not available on CRAN, its most recent version can be downloaded from the developers' website at http://www.r-tutor.com/content/download.

The http://www.r-tutor.com/gpu-computing site contains a number of practical applications of GPU-accelerated functions included in the rpud package.

Also, the NVIDIA CUDA Zone-a blog run by CUDA developers, presents very good tutorials on the implementation of selected rpud methods using R and Cloud computing (for example http://devblogs.nvidia.com/parallelforall/gpu-accelerated-r-cloud-terap roc-cluster-service/).

4 Multi-threading with Microsoft R Open distribution

The acquisition of Revolution Analytics by Microsoft in summer 2015 sent a clear signal to the R community that the famous Redmond-based tech giant was soon going to re-package the already good and Big-Data-friendly Revolution R Open (RRO) distribution and enhance it by equipping it with more powerful capabilities. When writing this chapter, the Microsoft R Open (MRO) distribution, based on the previous RRO version, is only a few days old, but it has already energized R users. Unfortunately, it's too new to be incorporated into this book, as it requires quite extensive testing to assess the validity of Microsoft's claims in terms of MRO's performance.

According to MRO's developers, Microsoft R Open provides access to the multi-threaded Math Kernel Library (MKL) giving R computations an impressive boost across a spectrum of mathematical and statistical calculations.

Diagrams and comparisons of performance benchmarks available at https://mran.revolutionanalytics.com/documents/rro/multithread/ clearly indicate that MRO with as little as 1 core can significantly increase computation speed for a variety of operations. MRO equipped with four cores may make them run up to 48 times faster (for a matrix multiplication) compared with the R distribution obtained from CRAN.

Depending on the type of algorithm used during performance testing, the Microsoft RO distribution excelled in two areas of matrix calculation and matrix functions; these are where MRO recorded the greatest performance gains. The programming functions designed for loops, recursions, or control flow generally performed at the same speed as in CRAN R.

Microsoft R Open is supported on 64-bit platforms only including Windows 7.X, Linux (Ubuntu, CentOS, Red Hat, SUS, and others.), and Mac OS X (10.9+), and can be installed from https://mran.revolutionanalytics.com/download/.

4.1 Parallel machine learning with H2O and R

In this section we will only very briefly mention the mere existence of H2O (http://www.h2o.ai/)-a fast and scalable platform for parallel and Big Data machine learning algorithms.

The platform is also supported by the R language through the h2o package (authored by Aiello, Kraljevic, and Maj with contributions from the actual developers of the H2O.aiteam), which provides an interface for the H2O open-source ML engine. This exciting collaboration is only mentioned here as we dedicate a large part of Chapter 8, Machine Learning methods for Big Data in R to a detailed discussion and a number of practical tutorials of the H2O platform and R applied to a real-world, Big Data issue.

4.2 Boosting R performance with the data.table package and other tools

The following two sections present several methods of enhancing the speed of data processing in R.

The larger part is devoted to the excellent data.table package, which allows convenient and fast data transformations. At the very end of this section we also direct you to other sources, that elaborate, in more detail, on the particulars of faster and better optimized R code.

4.3 Fast data import and manipulation with the data.table package

In a chapter devoted to optimized and faster data processing in the R environment, we simply must spare a few pages for one, extremely efficient and flexible package called data.table. The package, developed by Dowle, Srinivasan, Short, and Lianoglou with further contributions from Saporta and Antonyan, took the primitive R data.frame concept one (huge) step forward and has made the lives of many R users so much easier since its release to the community.

The data.table library offers (very) fast subsetting, joins, data transformations, and aggregations as well as enhanced support for fast date extraction and data file import.

But this is not all, it also has other great selling points, for example:

- A very convenient and easy chaining of operations
- Key setting functionality allowing (even!) faster aggregations
- Smooth transition between data.table and data.frame (if it can't find a data.table function it uses the base R expression and applies it on a data.frame, so users don't have to convert between data structures explicitly) -`Really easy-to-learn, natural syntax.

Oh, did I say it's fast? How about any drawbacks? It still stores data and all created data.table objects in RAM, but owing to its better memory management, it engages RAM only for processes, and only on specific subsets (rows, columns) of the data that have to be manipulated. The truth is that data.table can in fact save you a bit of cash if you work with large datasets.

As its computing time is much shorter than when using standard base R functions on data frames, it will significantly reduce your time and bill for cloudcomputing solutions.

But instead of reading about its features, why don't you try to experience them first hand by following the introductory tutorial?

4.4 Data import with data.table

In order to show you real performance gains with data.table we will be using the flight dataset , which we have already explored when discussing the ff/ffdf approach at the beginning of this chapter.

As you probably remember, we were comparing the speed of processing and memory consumption between the ff/ffbase packages and base R functions such as read.table() or read.csv() performed on a smaller dataset with two months of flights (you can download this from Packt Publishing's website for this book), and a bigger, almost 2 GB two-year dataset for which we were also giving performance benchmarks just as a reference.

In this section, we will be quoting only the estimates for the larger, 2 GB file with 12,189,293 observations, but feel free to follow the examples by running the same code on smaller data (just remember to specify the name of the file correctly).

The data.table package imports the data through its fast file reader fread() function:

```
library(data.table)

## data.table 1.14.4 using 4 threads (see ?getDTthreads).  Latest news: r-
datatable.com

##
## Attaching package: 'data.table'

## The following object is masked from 'package:tictoc':
##
##     shift

## The following object is masked from 'package:bit':
##
##     setattr

flightsDT <- fread("flights_1314.txt", stringsAsFactors = TRUE)
```

It's quite spectacular that operation which took 456 seconds using read.table.ffdf() and 441 seconds with read.table() was slashed to a mere 29 seconds in fread().

The fread() function also comes with a large number of additional arguments users can set; for example, they may select or drop specific columns, define standard separators between read columns (sep) or even within columns (sep2), specify the number of rows to read (nrows), the number of rows to skip (skip), define column classes (colClasses), and many others.

The resulting object is both a data.table and a data.frame allowing the flexibility of syntax and applications depending on users' specific needs:

```
str(flightsDT)

## Classes 'data.table' and 'data.frame':   12189293 obs. of  28 variables:
##  $ YEAR              : int  2013 2013 2013 2013 2013 2013 2013 2013 2013
```

```
2013 ...
##  $ MONTH            : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DAY_OF_MONTH     : int  17 18 19 20 21 22 23 24 25 26 ...
##  $ DAY_OF_WEEK      : int  4 5 6 7 1 2 3 4 5 6 ...
##  $ FL_DATE          : IDate, format: "2013-01-17" "2013-01-18" ...
##  $ UNIQUE_CARRIER   : Factor w/ 16 levels "9E","AA","AS",..: 1 1 1 1 1 1 1
1 1 1 ...
##  $ AIRLINE_ID       : int  20363 20363 20363 20363 20363 20363 20363 20363
20363 20363 ...
##  $ TAIL_NUM         : Factor w/ 5220 levels "","D942DN","N001AA",..: 4807
4659 4724 4787 4939 2759 4750 4930 4659 4607 ...
##  $ FL_NUM           : int  3324 3324 3324 3324 3324 3324 3324 3324 3324
3324 ...
##  $ ORIGIN_AIRPORT_ID: int  11298 11298 11298 11298 11298 11298 11298 11298
11298 11298 ...
##  $ ORIGIN           : Factor w/ 334 levels "ABE","ABI","ABQ",..: 92 92 92
92 92 92 92 92 92 92 ...
##  $ ORIGIN_CITY_NAME : Factor w/ 329 levels "Aberdeen, SD",..: 78 78 78 78
78 78 78 78 78 78 ...
##  $ ORIGIN_STATE_NM  : Factor w/ 53 levels "Alabama","Alaska",..: 44 44 44
44 44 44 44 44 44 44 ...
##  $ ORIGIN_WAC       : int  74 74 74 74 74 74 74 74 74 74 ...
##  $ DEST_AIRPORT_ID  : int  12478 12478 12478 12478 12478 12478 12478 12478
12478 12478 ...
##  $ DEST             : Factor w/ 332 levels "ABE","ABI","ABQ",..: 174 174
174 174 174 174 174 174 174 174 ...
##  $ DEST_CITY_NAME   : Factor w/ 327 levels "Aberdeen, SD",..: 220 220 220
220 220 220 220 220 220 220 ...
##  $ DEST_STATE_NM    : Factor w/ 53 levels "Alabama","Alaska",..: 32 32 32
32 32 32 32 32 32 32 ...
##  $ DEST_WAC         : int  22 22 22 22 22 22 22 22 22 22 ...
##  $ DEP_TIME         : int  1038 1037 1035 1037 1044 1054 1036 1036 1042
1034 ...
##  $ DEP_DELAY        : int  -7 -8 -5 -8 -1 9 -9 -9 -3 -6 ...
##  $ ARR_TIME         : int  1451 1459 1515 1455 1446 1502 1504 1520 1525
1509 ...
##  $ ARR_DELAY        : int  -14 -6 17 -10 -19 -3 -1 15 20 11 ...
##  $ CANCELLED        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ CANCELLATION_CODE: Factor w/ 5 levels "","A","B","C",..: 1 1 1 1 1 1 1
1 1 1 ...
##  $ DIVERTED         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ AIR_TIME         : int  175 178 201 176 162 171 186 204 174 189 ...
##  $ DISTANCE         : int  1391 1391 1391 1391 1391 1391 1391 1391 1391
1391 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

This flexibility of data.table semantics is best noticed in data transformations such as subsetting and aggregations.

4.5 Lightning-fast subsets and aggregations on data.table

Datatables can be subsetted and aggregated using their indexing operators surrounded by square [] brackets and with the following default format:

DT[i, j, by]

The structure of this call can be compared to a standard SQL query where the i operator stands for WHERE, j denotes SELECT, and by can be simply translated to the SQL GROUPBY statement.

In the most basic form we may subset specific rows, which match set conditions as in the example below:

```
subset1.DT <- flightsDT[ YEAR == 2013L & DEP_TIME >= 1200L & DEP_TIME <
1700L,]
str(subset1.DT)

## Classes 'data.table' and 'data.frame':   1933463 obs. of  28 variables:
##  $ YEAR             : int  2013 2013 2013 2013 2013 2013 2013 2013 2013
2013 ...
##  $ MONTH            : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DAY_OF_MONTH     : int  30 3 4 5 6 7 8 9 10 11 ...
##  $ DAY_OF_WEEK      : int  3 4 5 6 7 1 2 3 4 5 ...
##  $ FL_DATE          : IDate, format: "2013-01-30" "2013-01-03" ...
##  $ UNIQUE_CARRIER   : Factor w/ 16 levels "9E","AA","AS",..: 1 1 1 1 1 1 1
1 1 1 ...
##  $ AIRLINE_ID       : int  20363 20363 20363 20363 20363 20363 20363 20363
20363 20363 ...
##  $ TAIL_NUM         : Factor w/ 5220 levels "","D942DN","N001AA",..: 4750
4691 4637 4607 4724 4826 4659 2778 4769 4787 ...
##  $ FL_NUM           : int  3324 3325 3325 3325 3325 3325 3325 3325 3325
3325 ...
##  $ ORIGIN_AIRPORT_ID: int  11298 12478 12478 12478 12478 12478 12478 12478
12478 12478 ...
##  $ ORIGIN           : Factor w/ 334 levels "ABE","ABI","ABQ",..: 92 175
175 175 175 175 175 175 175 175 ...
##  $ ORIGIN_CITY_NAME : Factor w/ 329 levels "Aberdeen, SD",..: 78 222 222
222 222 222 222 222 222 222 ...
##  $ ORIGIN_STATE_NM  : Factor w/ 53 levels "Alabama","Alaska",..: 44 32 32
32 32 32 32 32 32 32 ...
##  $ ORIGIN_WAC       : int  74 22 22 22 22 22 22 22 22 22 ...
##  $ DEST_AIRPORT_ID  : int  12478 11298 11298 11298 11298 11298 11298 11298
11298 11298 ...
##  $ DEST             : Factor w/ 332 levels "ABE","ABI","ABQ",..: 174 91 91
91 91 91 91 91 91 91 ...
##  $ DEST_CITY_NAME   : Factor w/ 327 levels "Aberdeen, SD",..: 220 77 77 77
77 77 77 77 77 77 ...
##  $ DEST_STATE_NM    : Factor w/ 53 levels "Alabama","Alaska",..: 32 44 44
44 44 44 44 44 44 44 ...
```

```
##  $ DEST_WAC         : int  22 74 74 74 74 74 74 74 74 74 ...
##  $ DEP_TIME         : int  1538 1617 1643 1610 1603 1606 1612 1605 1601
1614 ...
##  $ DEP_DELAY        : int  293 12 38 5 -2 1 7 0 -4 9 ...
##  $ ARR_TIME         : int  1953 1925 1926 1929 1916 1852 2025 1932 1856
1852 ...
##  $ ARR_DELAY        : int  288 0 1 5 -9 -33 NA 7 -29 -33 ...
##  $ CANCELLED        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ CANCELLATION_CODE: Factor w/ 5 levels "","A","B","C",..: 1 1 1 1 1 1 1
1 1 1 ...
##  $ DIVERTED         : int  0 0 0 0 0 0 1 0 0 0 ...
##  $ AIR_TIME         : int  167 220 208 234 209 196 NA 221 202 196 ...
##  $ DISTANCE         : int  1391 1391 1391 1391 1391 1391 1391 1391 1391
1391 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

The task took only 1.19 seconds to complete, and the new subset (a data.table and a data.frame) contains all 2,013 flights which departed in the afternoon between 12:00 and 16:59.

In the same way we can perform simple aggregations in which we may even calculate other arbitrary statistics.

In the following example we will estimate the total delay (TotDelay) for each December flight and the average departure delay (AvgDepDelay) for all December flights. Additionally, we will group the results by the state of the flight origin (ORIGIN_STATE_NM):

```
subset2.DT <- flightsDT[ MONTH == 12L, .(TotDelay = ARR_DELAY - DEP_DELAY,
AvgDepDelay = mean(DEP_DELAY, na.rm = TRUE)), by = .(ORIGIN_STATE_NM) ]
subset2.DT

##         ORIGIN_STATE_NM TotDelay AvgDepDelay
##      1:        New York        1    11.85232
##      2:        New York      -42    11.85232
##      3:        New York      -16    11.85232
##      4:        New York      -32    11.85232
##      5:        New York       -2    11.85232
##     ---
## 993918:        Delaware       11    11.73494
## 993919:        Delaware       -3    11.73494
## 993920:        Delaware        0    11.73494
## 993921:        Delaware       -5    11.73494
## 993922:        Delaware        4    11.73494
```

By indicating the names of columns in the j parameter you can easily extract variables of interest, for example:

```
subset3.DT <- flightsDT[, .(MONTH, DEST)]
str(subset3.DT)
```

```
## Classes 'data.table' and 'data.frame':   12189293 obs. of  2 variables:
##  $ MONTH: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DEST : Factor w/ 332 levels "ABE","ABI","ABQ",..: 174 174 174 174 174
174 174 174 174 174 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

As in the previous listing, we may now quickly aggregate any statistic in j by group, specified in the by operator:

```
agg1.DT <- flightsDT[, .(SumCancel = sum(CANCELLED), MeanArrDelay =
mean(ARR_DELAY, na.rm = TRUE)), by = .(ORIGIN_CITY_NAME)]
agg1.DT

##            ORIGIN_CITY_NAME SumCancel MeanArrDelay
##   1:  Dallas/Fort Worth, TX     13980   10.0953618
##   2:           New York, NY     12264    6.0086474
##   3:        Minneapolis, MN      2976    3.6519174
##   4:     Raleigh/Durham, NC      2082    5.8777458
##   5:           Billings, MT        75   -1.0170240
##  ---
## 325:        Devils Lake, ND        10   13.6372240
## 326:            Hyannis, MA         1   -0.6933333
## 327:               Hays, KS        16   -3.0204082
## 328:           Meridian, MS         3    9.8698630
## 329: Hattiesburg/Laurel, MS         2   10.0434783
```

In the preceding snippet we simply calculated the number of cancelled flights, which were supposed to depart from each city, and the mean arrival delay for all remaining connections, which flew from specific locations.

The resulting data.table may be sorted using order() just as in base R. However, the data.table package offers an internally-optimized implementation of the order() function, which performs much faster on large datasets than its generic counterpart. We will now compare both implementations by sorting arrival delay values (ARR_DELAY) in the decreasing order for all flights in our data (12,189, 293 observations):

```
system.time(flightsDT[base::order(-ARR_DELAY)])

##    user  system elapsed
##    8.81    1.06    3.52

system.time(flightsDT[order(-ARR_DELAY)])

##    user  system elapsed
##    7.80    0.66    2.67
```

The data.table implementation is clearly at least 3x faster than when R was forced to use the base order() function.

The package contains a number of other shortcuts that speed up data processing; for example .N can be used to produce fast frequency calculations:

```
agg2.DT <- flightsDT[, .N, by = ORIGIN_STATE_NM]
agg2.DT
```

```
##                           ORIGIN_STATE_NM       N
##  1:                                 Texas 1463283
##  2:                              New York  553855
##  3:                             Minnesota  268206
##  4:                        North Carolina  390979
##  5:                               Montana   34372
##  6:                                  Utah  227066
##  7:                              Virginia  356462
##  8:                              Michigan  332694
##  9:                             Tennessee  197671
## 10:                              Missouri  224081
## 11:                             Louisiana  146652
## 12:                         Massachusetts  224082
## 13:                              Kentucky  109105
## 14:                           Connecticut   44041
## 15:                              Illinois  806230
## 16:                               Florida  871200
## 17:                                  Iowa   44197
## 18:                               Indiana   87487
## 19:                          Pennsylvania  238888
## 20:                              Colorado  498276
## 21:                          North Dakota   34352
## 22:                              Maryland  191182
## 23:                            Washington  238145
## 24:                             Wisconsin  120927
## 25:                              Oklahoma   85725
## 26:                                  Ohio  178884
## 27:                               Georgia  802243
## 28:                              Nebraska   49068
## 29:                        South Carolina   67583
## 30:                          South Dakota   23645
## 31:                              Arkansas   59792
## 32:                                Kansas   26239
## 33:                          Rhode Island   27542
## 34:                         New Hampshire   16720
## 35:                               Alabama   66695
## 36:                               Vermont    9817
## 37:                            New Jersey  235960
## 38:                                 Maine   13795
## 39:                           Mississippi   29009
## 40:                                Nevada  310313
## 41:                            California 1495110
## 42:                                Hawaii  204652
## 43:                               Arizona  383253
## 44:                           Puerto Rico   56062
## 45:                            New Mexico   60354
## 46:                   U.S. Virgin Islands    9869
```

```
## 47:                                          Oregon  132502
## 48:                                          Alaska   74589
## 49:                                         Wyoming   21144
## 50:                                           Idaho   36449
## 51:                                   West Virginia    6840
## 52: U.S. Pacific Trust Territories and Possessions     951
## 53:                                        Delaware    1055
##                                     ORIGIN_STATE_NM       N
```

R spent only 0.098 seconds estimating the counts of flights from each state. Compared to the table() approach on a data.frame (1.14 seconds), the data.table package offers roughly a ten-fold speedup:

```
agg2.df <- as.data.frame(table(flightsDT$ORIGIN_STATE_NM))
agg2.df
```

```
##                                              Var1    Freq
## 1                                         Alabama   66695
## 2                                          Alaska   74589
## 3                                         Arizona  383253
## 4                                        Arkansas   59792
## 5                                      California 1495110
## 6                                        Colorado  498276
## 7                                     Connecticut   44041
## 8                                        Delaware    1055
## 9                                         Florida  871200
## 10                                        Georgia  802243
## 11                                         Hawaii  204652
## 12                                          Idaho   36449
## 13                                       Illinois  806230
## 14                                        Indiana   87487
## 15                                           Iowa   44197
## 16                                         Kansas   26239
## 17                                       Kentucky  109105
## 18                                      Louisiana  146652
## 19                                          Maine   13795
## 20                                       Maryland  191182
## 21                                  Massachusetts  224082
## 22                                       Michigan  332694
## 23                                      Minnesota  268206
## 24                                    Mississippi   29009
## 25                                       Missouri  224081
## 26                                        Montana   34372
## 27                                       Nebraska   49068
## 28                                         Nevada  310313
## 29                                  New Hampshire   16720
## 30                                     New Jersey  235960
## 31                                     New Mexico   60354
## 32                                       New York  553855
## 33                                 North Carolina  390979
```

```
## 34                                   North Dakota   34352
## 35                                           Ohio  178884
## 36                                       Oklahoma   85725
## 37                                         Oregon  132502
## 38                                   Pennsylvania  238888
## 39                                    Puerto Rico   56062
## 40                                   Rhode Island   27542
## 41                                 South Carolina   67583
## 42                                   South Dakota   23645
## 43                                      Tennessee  197671
## 44                                          Texas 1463283
## 45 U.S. Pacific Trust Territories and Possessions     951
## 46                            U.S. Virgin Islands    9869
## 47                                           Utah  227066
## 48                                        Vermont    9817
## 49                                       Virginia  356462
## 50                                     Washington  238145
## 51                                  West Virginia    6840
## 52                                      Wisconsin  120927
## 53                                        Wyoming   21144
```

4.6 Chaining, more complex aggregations, and pivot tables with data.table

One of the smartest things about data.table is that users can easily chain several fast operations into one expression reducing both programming and computing time, forexample:

```
agg3.DT <- flightsDT[, .N, by = ORIGIN_STATE_NM] [order(-N)]
agg3.DT
```

```
##                                 ORIGIN_STATE_NM       N
##  1:                                  California 1495110
##  2:                                       Texas 1463283
##  3:                                     Florida  871200
##  4:                                    Illinois  806230
##  5:                                     Georgia  802243
##  6:                                    New York  553855
##  7:                                    Colorado  498276
##  8:                              North Carolina  390979
##  9:                                     Arizona  383253
## 10:                                    Virginia  356462
## 11:                                    Michigan  332694
## 12:                                      Nevada  310313
## 13:                                   Minnesota  268206
## 14:                                Pennsylvania  238888
## 15:                                  Washington  238145
## 16:                                  New Jersey  235960
## 17:                                        Utah  227066
## 18:                               Massachusetts  224082
## 19:                                    Missouri  224081
```

```
## 20:                                        Hawaii  204652
## 21:                                     Tennessee  197671
## 22:                                      Maryland  191182
## 23:                                          Ohio  178884
## 24:                                     Louisiana  146652
## 25:                                        Oregon  132502
## 26:                                     Wisconsin  120927
## 27:                                      Kentucky  109105
## 28:                                       Indiana   87487
## 29:                                      Oklahoma   85725
## 30:                                        Alaska   74589
## 31:                                South Carolina   67583
## 32:                                       Alabama   66695
## 33:                                    New Mexico   60354
## 34:                                      Arkansas   59792
## 35:                                   Puerto Rico   56062
## 36:                                      Nebraska   49068
## 37:                                          Iowa   44197
## 38:                                   Connecticut   44041
## 39:                                         Idaho   36449
## 40:                                       Montana   34372
## 41:                                  North Dakota   34352
## 42:                                   Mississippi   29009
## 43:                                  Rhode Island   27542
## 44:                                        Kansas   26239
## 45:                                  South Dakota   23645
## 46:                                       Wyoming   21144
## 47:                                 New Hampshire   16720
## 48:                                         Maine   13795
## 49:                           U.S. Virgin Islands    9869
## 50:                                       Vermont    9817
## 51:                                 West Virginia    6840
## 52:                                      Delaware    1055
## 53: U.S. Pacific Trust Territories and Possessions     951
##                                   ORIGIN_STATE_NM       N
```

Chaining is especially useful in more complex aggregations.

In the following example we want to calculate the arithmetic mean (set in the j index) for all December flights (i index) on departure and arrival delay variables as indicated by the .SDcols parameter, group the results by the ORIGIN_STATE_NM, DEST_STATE_NM, and DAY_OF_WEEK variables, and finally sort the output firstly by DAY_OF_WEEK (in ascending order) and then by DEP_DELAY and ARR_DELAY (both in descending order):

```
agg4.DT <- flightsDT[MONTH == 12L,
                     lapply(.SD, mean, na.rm = TRUE),
                     by = .(ORIGIN_STATE_NM, DEST_STATE_NM,DAY_OF_WEEK),
                     .SDcols = c("DEP_DELAY", "ARR_DELAY")]
[order(DAY_OF_WEEK, -DEP_DELAY, -ARR_DELAY)]
head(agg4.DT, n=5)
```

```
##    ORIGIN_STATE_NM       DEST_STATE_NM DAY_OF_WEEK DEP_DELAY ARR_DELAY
## 1:       Louisiana            Kentucky           1  111.6667  108.0000
## 2:            Ohio      South Carolina           1  106.0000  104.3333
## 3:          Alaska               Texas           1  103.0000   93.0000
## 4:    Pennsylvania U.S. Virgin Islands           1   92.0000   88.5000
## 5:         Indiana           Tennessee           1   90.0000   82.7500
```

Note that the expression looks very tidy and contains multiple data-manipulation techniques.

It is also extremely fast even on a large dataset; the preceding statement was executed in only 0.13 seconds.

In order to replicate this aggregation in the base R, we would probably need to create a separate function, to process the call in stages, but it is very likely that the performance of this approach will be much worse.

The data.table package through elegant chaining allows users to shift their focus from programming to true data science.The chaining of operations may also be achieved through custom-built functions.

In the following example, we want to create a delay function that will calculate TOT_DELAY for each flight in the dataset and we also want to attach this variable to our main dataset using the := operator. Second, based on the newly-created TOT_DELAY variable, the function will compute MEAN_DELAY for each DAY_OF_MONTH:

```
delay <- function(DT) {
  DT[, TOT_DELAY := ARR_DELAY - DEP_DELAY]
  DT[, .(MEAN_DELAY = mean(TOT_DELAY, na.rm = TRUE)), by = DAY_OF_MONTH]
}
delay.DT <- delay(flightsDT)
names(flightsDT)
```

```
##  [1] "YEAR"              "MONTH"             "DAY_OF_MONTH"
##  [4] "DAY_OF_WEEK"       "FL_DATE"           "UNIQUE_CARRIER"
##  [7] "AIRLINE_ID"        "TAIL_NUM"          "FL_NUM"
## [10] "ORIGIN_AIRPORT_ID" "ORIGIN"            "ORIGIN_CITY_NAME"
## [13] "ORIGIN_STATE_NM"   "ORIGIN_WAC"        "DEST_AIRPORT_ID"
## [16] "DEST"              "DEST_CITY_NAME"    "DEST_STATE_NM"
## [19] "DEST_WAC"          "DEP_TIME"          "DEP_DELAY"
## [22] "ARR_TIME"          "ARR_DELAY"         "CANCELLED"
## [25] "CANCELLATION_CODE" "DIVERTED"          "AIR_TIME"
## [28] "DISTANCE"          "TOT_DELAY"
```

```
head(delay.DT)
```

```
##    DAY_OF_MONTH MEAN_DELAY
## 1:           17  -3.235925
## 2:           18  -3.369053
## 3:           19  -3.439632
## 4:           20  -3.976177
```

```
## 5:            21  -3.229061
## 6:            22  -3.166394
```

The := operator added the TOT_DELAY variable to the original data stored in the flightsDT object and the delay function computed the requested MEAN_DELAY by DAY_OF_MONTH and stored the results in delay.DT of the data.table package.

We should also mention here a very useful casting implementation through the dcast.data.table() function which allows rapid pivot tables, for example:

```
agg5.DT <- dcast.data.table(flightsDT,
                            UNIQUE_CARRIER~MONTH,
                            fun.aggregate = mean,
                            value.var = "TOT_DELAY",
                            na.rm=TRUE)
agg5.DT
```

```
##     UNIQUE_CARRIER          1          2          3           4          5
##  1:             9E -5.0604885 -4.3035723 -3.9639291 -3.97549369 -3.9851160
##  2:             AA -5.4185064 -4.7585774 -4.8034317 -3.52661907 -3.9940437
##  3:             AS -3.6258772 -3.9423103 -2.5658019 -1.42202236 -1.1287699
##  4:             B6 -3.0075933 -1.6200692 -2.9693669 -2.58131086 -4.7723310
##  5:             DL -4.9593914 -5.1271330 -4.9993318 -4.78600081 -4.5556703
##  6:             EV -2.8316765 -2.6735762 -3.5791400 -3.26905250 -3.3592182
##  7:             F9  1.5128138  1.7389582  1.0289684  1.64470449  0.8981428
##  8:             FL -3.4655201 -2.9572431 -2.2638647 -3.41077019 -4.5667835
##  9:             HA  0.9881499  0.5728666  0.9751099  0.87703793  0.9442306
## 10:             MQ -0.2217426 -0.2059641 -1.1987497  0.05964314 -0.1753046
## 11:             OO -1.1938984 -1.2106156 -1.8446472 -1.58304790 -1.7056795
## 12:             UA -7.5894200 -6.7246802 -7.8630617 -7.08422261 -7.4097555
## 13:             US -2.0626065 -1.3832280 -2.3973111 -1.80466899 -2.0518061
## 14:             VX -6.5903668 -4.8921029 -5.7629638 -5.48504524 -5.1271590
## 15:             WN -5.6655179 -5.9149595 -4.9892291 -4.59273465 -5.0319650
## 16:             YV -1.1544345 -0.8871053 -1.5168986 -0.48641007 -0.9311323
##              6           7           8          9          10         11
##  1: -2.6049019 -3.07248526 -4.65300916 -4.9229329 -4.61331220 -5.5528514
##  2: -2.4831712 -2.67815547 -3.59220357 -4.1196392 -2.80719209 -4.6130687
##  3: -0.3347246  0.13715137  0.20431347 -0.7381192 -1.75437987 -3.8927848
##  4: -3.1121345 -1.03585986 -2.78829268 -3.6394169 -3.41829733 -3.6533926
##  5: -3.6840302 -4.58935183 -5.20531640 -4.3365582 -5.11543072 -4.9731944
##  6: -2.5741488 -3.02601317 -3.36049714 -2.8986387 -2.19604016 -3.1766812
##  7:  1.3757110  1.33186798  0.70284411  0.5010832 -0.58000545  0.8790109
##  8: -2.4149313 -3.12670845 -4.48336511 -5.2597094 -4.52553144 -5.4973022
##  9:  0.4227802  0.98846297  0.84973424  1.0794127  1.47679909  1.7353040
## 10:  0.9934712 -0.07574324 -0.15533339 -1.6402856 -0.09883242 -0.5689987
## 11: -1.2013261 -1.46349066 -1.13168249 -1.1756191 -1.09401480 -1.6460572
## 12: -5.6482530 -6.81243984 -7.14127087 -7.0770797 -6.86042483 -8.2813032
## 13: -0.2648810 -0.48139653 -1.78743177 -2.8849762 -2.12301615 -2.9136960
## 14: -2.7562313 -4.56795741 -4.55636039 -4.5778543 -5.33036085 -4.7851608
## 15: -3.9938371 -4.23416676 -4.19607188 -4.4103769 -5.33155108 -6.3515294
```

```
## 16:  0.5402868  0.75983760  0.01435857 -0.3753379 -0.01849384 -0.3619767
##             12
##  1: -1.2206257
##  2: -3.8879515
##  3: -3.3385013
##  4: -2.1226683
##  5: -6.0661009
##  6: -1.9626498
##  7:  1.2813819
##  8: -3.2641951
##  9:  1.3017804
## 10:  1.0469016
## 11: -0.1364558
## 12: -7.0841910
## 13: -2.0027092
## 14: -4.1633648
## 15: -5.1899477
## 16:  0.3937927
```

In the preceding output we have obtained a pivot table with mean values of TOT_DELAY for each carrier and month in our data.

The fun.aggregate argument may take multiple functions, and similarly, the value.var parameter may now also refer to multiple columns.

In this section we have presented several common applications of fast data transformations available in the data.table package.

This is, however, not inclusive of all functionalities this great package can offer. For more examples (especially on chaining, joins, key setting, and many others) and tutorials, please visit the data.table GitHub repository at https://github.com/Rdatatable/data.table/wiki. The CRAN page for the package (https://cran.r-project.org/web/packages/data.table/index.html) contains several references to comprehensive manuals elaborating on different aspects of data manipulation with data.table.

4.7 Writing better R code

Finally, in the last section of this chapter we will direct you to several good sources that can assist you in writing better optimised and faster R code.

The best primary resource of knowledge on this subject is the previously mentioned book by Hadley Wickham Advanced R, and more specifically its chapter on code optimization. It includes very informative, but still pretty concise and approachable, sections on profiling and benchmarking tools, which can be used to test the performance of R scripts. Wickham also shares a number of tips and tricks on how to organize the code, minimise the workload, compile the functions, and use other techniques such as R interfaces for compiled code. The online version of the chapter is available from Wickham's personal website at http://adv-r.had.co.nz/Profiling.html.

Another great source of information in this field is Norman Matloff's book titled The Art of R Programming. It consists of several comprehensive chapters dedicated to performance enhancements in processing speed and memory consumption, interfacing R to other languages-most predominantly C/C++ and Python, and also introducing readers to parallel techniques available in R through Hadley Wickham's snow package, compiled code and GPU programming amongst others. Besides, Matloff includes essential details on code debugging and rectifying issues with specific programming methods in R.

Unfortunately, the code optimization goes beyond the scope of this book, but the contents of both Wickham's and Matloff's publications cover this gap in a very comprehensive way.

There are also a number of good web-based resources on specific high-performance computing approaches available in R and the CRAN Task View https://cran.r-project.org/web/views/HighPerformanceComputing.html should serve well as the index of the most essential packages and tools on that subject. Also, most of the packages, that were either referenced in the preceding sections of this chapter, or are listed in the High-Performance Computing Task View on CRAN, contain wellwritten manuals, and at least vignettes, addressing most important concepts and common applications.

4.8 Summary

In this chapter we have began our journey through the meanders of Big Data analytics with R.

First, we introduced you to the structure, definition, and major limitations of the R programming language hoping that this may clarify why traditionally R was an unlikely choice for a Big Data analyst.

But then we showed you how some of these concerns can be quite easily dispelled by using several powerful R packages which facilitate processing and analysis of large datasets.

We have spent a large proportion of this chapter on approaches, that allow out-of-memory data management, first through the ff and ffbase packages, and later by presenting methods contained within the bigmemory package and other libraries that support operations and analytics on big.matrix objects.

In the second part of the chapter we moved on to methods that can potentially boost the performance of your R code.

We explored several applications of parallel computing through the parallel and foreach packages and you learnt how to calculate statistics using the apply() family of functions and for() loops.

We also provided you with a gentle introduction to GPU computing, a new highly-optimized Microsoft R Open distribution, which supports multi-threading, and we have also mentioned the H2O platform for fast and scalable machine learning for Big Data (which we discuss in detail in Chapter 8, Machine Learning methods for Big Data in R).

We ended this chapter with an introductory tutorial on the data.table package-a highly efficient and popular package for fast data manipulation in R.

Further if you want to know how we can take Big Data outside of the limitation of a single machine and to deploy and configure instances and clusters on leading Cloud computing platforms, such as Amazon Elastic Cloud Computing (EC2), Microsoft Azure, and Google Cloud, you can go through the Online Chapter, Pushing R Further available at h t t p s : / / w w w . p a c k t p u b . c o m / s i t e s / d e f a u l t / f i l e s / d o w n l o a d s / 5 3 9 6 6 4 5 7 O S P u s h i n g R F u r t h e r . p d f