

# partools: a Sensible Package for Large Data Sets

Norm Matloff, with Contributions from Alex Rumbaugh

July 15, 2015

With the advent of Big Data, the Hadoop framework became ubiquitous. Yet it was clear that Hadoop had major shortcomings, and recently these are being much more seriously discussed.<sup>1</sup> This has resulted in a new platform, Spark, gaining popularity. As with Hadoop, there is an R interface available for Spark, named SparkR.

Spark overcomes one of Hadoop's major problems, which is the lack of ability to cache data in a multi-pass computation. However, Spark unfortunately retains the drawbacks of Hadoop:

- Hadoop/Spark have a complex, rather opaque infrastructure, and rely on Java/Scala. This makes them difficult to install, configure and use for those who are not computer systems experts.
- Although a major plus for Hadoop/Spark is fault tolerance, it is needed only for users working on extremely large clusters, consisting of hundreds or thousands of nodes. Disk failure rates are simply too low for fault tolerance to be an issue for many Hadoop/Snow users, who do not have such large systems.<sup>2</sup> Worse, the fault tolerance mechanisms, which are especially extensive in Spark, *slow down the computation*.

The one firm advantage of Hadoop/Spark is their use of distributed file systems. Under the philosophy, “Move the computation to the data, rather than *vice versa*,” network traffic may be greatly reduced, thus speeding up computation.

Therefore:

It is desirable to have a package that retains the distributed-file nature of Hadoop/Spark while staying fully within the simple, familiar, yet powerful R framework.

The **partools** package is designed to meet these goals. It is intended as a **simple, sensible alternative to Snow/Hadoop**. Though not necessarily appropriate for all settings, for many R programmers, **partools** may be a much better choice than **Hadoop/Snow**.

Since **partools** uses the portion of the R **parallel** package derived from the package **snow**, and because it is meant as an alternative to Hadoop, we informally refer to **partools** as Snowdoop.

The package does not provide fault tolerance of its own. If this is an issue, one can provide it externally, say with the XtremFS system.

---

<sup>1</sup>See for example “The Hadoop Honeymoon is Over,” <https://www.linkedin.com/pulse/hadoop-honeymoon-over-martyn-jones>

<sup>2</sup><https://wiki.apache.org/hadoop/PoweredBy>

# 1 Where Is the Magic?

As you will see later, **partools** can deliver some impressive speedups. But there is nothing magical about this. Instead, the value of the package consists of just two simple sources:

- (a) The package follows a philosophy of forming distributed objects *and keeping using them in distributed form throughout one's R session*.
- (b) The package consists of a number of utility functions that greatly facilitate creating, storing and *analyzing* distributed objects.

## 2 Overview of the partools Package

The package is based on the following very simple principles, involving *distributed files* and *distributed data frames/matrices*.

- Files are stored in a distributed manner, in files with a common basename. For example, the file **x** is stored as separate files **x.01**, **x.02** etc.
- Data frames and matrices are stored in memory at the nodes in a distributed manner, with a common name. For example, the data frame **y** is stored in chunks at the cluster nodes, each chunk known as **y** at its node.

### 2.1 Package Structure

Again, in a distributed file, all the file chunks have the same prefix, and in a distributed data frame, all chunks have the same name at the various cluster nodes. This plays a key role in the software.

The package consists of three main groups of functions:

#### Distributed-file functions:

- **filesplit()**: Create a distributed file from monolithic one.
- **filesplitrand()**: Create a distributed file from monolithic one, but randomize the record order.
- **filecat()**: Create a monolithic file from distributed one.
- **fileread()**: Read a distributed file into distributed data frame.
- **readnscramble()**: Read a distributed file into distributed data frame, but randomize the record order.
- **filesave()**: Write a distributed data frame to a distributed file.
- **filechunkname()**: For the calling cluster node, returns the full name of the file chunk, including suffix, e.g. '01', '02' etc.

#### Tabulative functions:

- **distribsplit()**: Create a distributed data frame/matrix from monolithic one.
- **distribcat()**: Create a monolithic data frame/matrix from distributed one.
- **distribagg()**: Distributed form of R's **aggregate()**.
- **distribcounts()**: Wrapper for **distribagg()** to obtain cell counts.
- **distribgetrows()**: Applies an R **subset()** or similar filtering operation to the distributed object, and collects the results into a single object at the caller.
- **distribrange()**: Distributed form of R's **range()**.

### Statistical functions:

These all use the Software Alchemy (SA) method (*Parallel Computation for Data Science*, N. Matloff, Chapman and Hall, 2015) to parallelize statistical operations. The idea is simple: Apply the given estimator to each chunk in the distributed object, and average over chunks. It is proven that the resulting distributed estimator has the same statistical accuracy — the same asymptotic variance — as the original serial ones.<sup>3</sup>

- **ca()**: General SA algorithm.
- **cabase()**: Core of **ca()**.
- **caagg()**: SA analog of **distribagg()**.
- **cameans()**: Finds means in the specified columns.
- **caquantile()**: Wrapper for SA version of R's **quantile()**.
- **calm()**: Wrapper for SA version of R's **lm()**.
- **caglm()**: Wrapper for SA version of R's **glm()**.
- **cakm()**: Wrapper for SA version of R's **kmeans()**.
- **caprcomp()**: Wrapper for SA version of R's **prcomp()**.

### Support functions:

- **formrowchunks()**: Form chunks of rows of a data frame/matrix.
- **matrixtolist()**: For a list of the rows or columns of a data frame or matrix.
- **addlists()**: “Add” two lists, meaning add values of elements of the same name, and copy the others.
- **dbb(), dbbmsg(), etc.:** Debugging aids.

---

<sup>3</sup>In the world of parallel computation, the standard word for nonparallel is *serial*.

### 3 Sample Session

Our data set, from <http://stat-computing.org/dataexpo/2009/the-data.html> consists of the well-known records of airline flight delay. For convenience, we'll just use the data for 2008, which consists of about 7 million records. This is large enough to illustrate speedup due to parallelism, but small enough that we won't have to wait really long amounts of time in our sample session here.

The session was run on a 16-core machine, with a 16-node **parallel** cluster. Note carefully, though, that we should not expect a 16-fold speedup. In the world of parallel computation, one usually gets of speedups of considerably less than  $n$  for a platform of  $n$  computational entities. Indeed, one is often saddened to find that the parallel version is actually *slower* than the serial one!

The file, **yr2008**, was first split into a distributed file, stored in **yr2008r.01,...,yr2008r.16**, using **filesplitrand()**, and then read into memory at the 16 cluster nodes using **fileread()**:

```
> filesplitrand(cls,'yr2008','yr2008r',2,header=TRUE,sep=",")
> fileread(cls,'yr2008r','yr2008',2,header=TRUE, sep=",")
```

The call to **filesplitrand()** splits the file as described above; since these files are permanent, we can skip this step in future R sessions involving this data (if the file doesn't change). The function **filesplitrand()** was used instead of **filesplit()** to construct the distributed file, in order to randomize the placement of the records of **yr2008** across cluster nodes. Random arrangement of the rows is required for SA.

In order to run timing comparisons, the full file was also read into memory at the cluster manager:

```
> yr2008 <- read.csv("yr2008")
```

The first operation run involved the package's distributed version of R's **aggregate()**. Here we want to tabulate departure delay, arrival delay and flight time, broken down into cells according to flight origin and destination. We'll find the maximum value in each cell.

```
> system.time(print(distribagg(cls, c("DepDelay","ArrDelay","AirTime"),
  c("Origin","Dest"),"yr2008", FUN="max"))))
...
5193   CDV   YAK      327      325      54
5194   JNU   YAK      317      308      77
5195   SLC   YKM      110      118     115
5196   IPL   YUM      162      163      26
...
      user  system elapsed
2.291    0.084   15.952
```

The serial version was much slower.

```
> system.time(print(aggregate(cbind(DepDelay,ArrDelay,AirTime) ~
  Origin+Dest,data=yr2008,FUN=max)))
...
```

```

5193   CDV   YAK      327      325      54
5194   JNU   YAK      317      308      77
5195   SLC   YKM      110      118     115
5196   IPL   YUM      162      163      26
...
      user  system elapsed
249.038    0.444  249.634

```

So, the results of **distribagg()** did indeed match those of **aggregate()**, but did so more than 15 times faster!

Remember, the key philosophy of **partools** is to create distributed objects and then keep using them repeatedly in distributed form. However, in some cases, we may wish to collect a distributed result into a monolithic object, especially if the result is small. This is done in the next example:

Say we wish to do a filter operation, extracting the data on all the Sunday evening flights, and collect it into one place. Here is the direct version:

```
> sundayeve <- with(yr2008,yr2008[DayOfWeek==1 & DepTime > 1800,])
```

This actually is not a time-consuming operation, but again, in typical **partools** use, we would only have the distributed version of **yr2008**. Here is how we would achieve the same effect from the distributed object:

```
> sundayeved <- distribgetrows(cls,'with(yr2008,yr2008[DayOfWeek==1 & DepTime > 1800,])')
```

As another example, say we are investigating data completeness. We may wish to flag all records having an inordinate number of NA values. As I first step, we may wish to add a column to our data frame, indicating how many NA values are in each row. If we did not have the advantage of distributed computation, here is how long it would take for our flight delay data:

```

> sumna
function(x) sum(is.na(x))
> system.time(yr2008$n1 <- apply(yr2008[,c(5,7,8,11:16,19:21)],1,sumna))
      user  system elapsed
268.463    0.773  269.542

```

But it is of course much faster on a distributed basis, using the **parallel** package function **clusterEvalQ()**:

```

> clusterExport(cls,"sumna",envir=environment())
> system.time(clusterEvalQ(cls,yr2008$n1 <- apply(yr2008[,c(5,7,8,11:16,19:21)],1,sumna)))
      user  system elapsed
  0.094    0.012   16.758

```

The speedup here was about 16, fully utilizing all 16 cores.

Ordinarily, we would continue that NA analysis on a distributed basis, in keeping with the **partools** philosophy of setting up distributed objects and then repeatedly dealing with them on a distributed

basis. If our subsequent operations continue to have time complexity linear in the number of records processes, we should continue to get speedups of about 16.

On the other hand, we may wish to gather together all the records have 8 or more NA values. In the nonparallel context, it would take some time:

```
> system.time(na8 <- yr2008[yr2008$n1 > 7,])
      user  system elapsed
9.292    0.028    9.327
```

In the distributed manner, it is slightly faster:

```
> system.time(na8d <- distribgetrows(cls,'yr2008[yr2008$n1 > 7,]'))
      user  system elapsed
5.524    0.160    6.584
```

The speedup is less here, as the resulting data must travel from the cluster nodes to the cluster manager. In this case, this is just a memory-to-memory transfer rather than across a network, as we are on a multicore machine, but it still takes time. If the number of records satisfying the filtering condition had been smaller than the 136246 we had here, the speedup factor would have been greater.

Now let's turn to statistical operations, starting of course with linear regression. As noted, these **partools** functions make use of Software Alchemy, which replaces the given operation by a *distributed, statistically equivalent* operation. This will often produce a significant speedup. Note again that though the result may differ from the non-distributed version, say in the third significant digit, it is just as accurate statistically.

In the flight data, we predicted the arrival delay from the departure delay and distance, comparing the distributed and serial versions,

```
> system.time(print(lm(ArrDelay ~ DepDelay+Distance,data=yr2008)))
...
Coefficients:
(Intercept)    DepDelay    Distance
-1.061369      1.019154    -0.001213

      user  system elapsed
77.107   12.463   76.225
> system.time(print(calm(cls,'ArrDelay ~ DepDelay+Distance,data=yr2008')$ttht))
(Intercept)    DepDelay    Distance
-1.061262941    1.019150592 -0.001213252
      user  system elapsed
13.414    0.691   18.396
```

Linear regression is very hard to parallelize, so the speedup of more than 4 here is nice. Coefficient estimates were virtually identical.

Next, principal components. Since R's **prcomp()** does not handle NA values for nonformula specifications, let's do that separately first:

```
> system.time(cc <- na.omit(yr2008[,c(12:16,19:21)]))
  user  system elapsed
 9.540   0.351   9.907
> system.time(clusterEvalQ(cls,cc <- na.omit(yr2008[,c(12:16,19:21)])))
  user  system elapsed
 0.885   0.232   2.352
```

Note that this too was faster in the distributed approach, though both times were small. And now the PCA runs:

```
> system.time(ccout <- prcomp(cc))
  user  system elapsed
61.905 49.605 58.444
> ccout$sdev
[1] 5.752546e+02 5.155227e+01 2.383117e+01 1.279210e+01 9.492825e+00
[6] 5.530152e+00 1.133015e-03 6.626621e-12
> system.time(ccoutdistr <- caprcomp(cls,'cc',8))
  user  system elapsed
 5.023   0.604   8.949
> ccoutdistr$sdev
[1] 5.752554e+02 5.155127e+01 2.383122e+01 1.279184e+01 9.492570e+00
[6] 5.529869e+00 9.933142e-04 8.679427e-13
```

Thus, more than a 6-fold speedup here. Agreement of the component standard deviations is good.

Next, let's find the *interquartile range* for several columns. This is a robust measure of dispersion, defined as the difference between the 75<sup>th</sup> and 25<sup>th</sup> percentiles. Here is serial code to find this:

```
# find the interquartile range for a vector x
iqr <- function(x) {tmp <- quantile(x,na.rm=T); tmp[4] - tmp[2]}
# find the interquartile range for each column of a data frame dfr
iqrM <- function(dfr) apply(dfr,2,iqr)
```

So, let's compare times. First, the serial version:

```
> system.time(print(iqrM(yr2008[,c(5:8,12:16,19:21)])))
      DepTime      CRSDepTime      ArrTime      CRSArrTime
      800          790          802          792
ActualElapsedTime CRSElapsedTime      AirTime      ArrDelay
      80          79          77          22
      DepDelay      Distance      TaxiIn      TaxiOut
      12          629          4          9
  user  system elapsed
29.280   0.243  29.554
```

For the distributed version,

```
> system.time(print(colMeans(distribgetrows(cls,'iqrM(yr2008[,c(5:8,12:16,19:21)]')'))))
```

	DepTime	CRSDepTime	ArrTime	CRSArrTime
	800.1250	790.0625	801.8125	791.8750
	ActualElapsedTime	CRSElapsedTime	AirTime	ArrDelay
	80.0000	78.9375	76.5625	22.0000
	DepDelay	Distance	TaxiIn	TaxiOut
	12.0000	627.6875	4.0000	9.0000
user	system	elapsed		
0.009	0.002	2.587		

Here the speedup was more than 11-fold, with agreement generally to three significant digits. Once again, note that statistically speaking, both estimators have the same accuracy.

The package also includes a distributed version of k-means clustering. Here it is on the flight delay data. First, retain only the NA-free rows for the variables of interest, then run:

```
> fileread(cls,'yr2008r','yr2008',2,header=TRUE, sep=",")
> invisible(clusterEvalQ(cls,y28 <- na.omit(yr2008[,c(5:8,13:16,19:21)])))
> system.time(koutpar <- cakm(cls,'y28',3,11))
  user  system elapsed
4.083   0.132   9.293
```

Compare to serial:

```
> yr2008 <- read.csv('y2008')
> y28 <- na.omit(yr2008[,c(5:8,13:16,19:21)])
> system.time(koutser <- kmeans(y28,3))
  user  system elapsed
54.394   0.558  55.032
```

So, the distributed version is about 6 times faster. Results are virtually identical:

```
> koutpar$centers
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 1741.3967 1718.0296 1876.433 1895.435 110.4398
[2,]  932.4057  936.6907 1081.743 1082.813 108.5091
[3,] 1311.2193 1308.1838 1496.267 1525.790 267.8620
      [,6]      [,7]      [,8]      [,9]     [,10]
[1,]  85.25672 13.101844 14.826940  569.2534  6.742315
[2,]  84.66763  3.091913  4.517502  561.0567  6.785464
[3,] 238.93152  8.668587 11.698193 1886.8964  7.541735
      [,11]
[1,] 16.71571
[2,] 15.63046
[3,] 18.35916
> koutser$centers
      DepTime CRSDepTime  ArrTime CRSArrTime
1 1741.3888 1718.0217 1876.418 1895.425
2  932.3681  936.6674 1081.737 1082.809
```



3	1311.4436	1308.3525	1496.404	1525.905
	CRSElapsedTime	AirTime	ArrDelay	DepDelay
1	110.4363	85.25292	13.100842	14.826083
2	108.5151	84.67361	3.092439	4.518112
3	267.8669	238.93668	8.672439	11.701706
	Distance	TaxiIn	TaxiOut	
1	569.2226	6.742079	16.71604	
2	561.1148	6.785409	15.63038	
3	1886.9094	7.542760	18.35823	