partools: a Sensible Package for Large Data Sets

Norm Matloff
University of California, Davis

February 5, 2026

The **partools** package is a general framework for parallel processing of large data sets and/or highly-computational algorithms. It is the "un-MapReduce," avoiding that paradigm while retaining the philosophy of highly-distributed memory objects and files of Hadoop and Spark.

1 Motivation

With the advent of Big Data, the Hadoop framework became ubiquitous in the years following its birth in 2006. Yet it was clear from the start that Hadoop had major shortcomings in performance, and eventually these came under serious discussion[1] This 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:

- Spark still relies on a MapReduce paradigm, featuring a *shuffle* (systemwide sort) operation after each pass of the data. This can be quite slow.[2]
- Thus SparkR can be considerably slower than Plain Old R (POR).
- 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 users who think they need Hadoop/Spark but who do not have such large systems.[3]

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.

[1] See for example "The Hadoop Honeymoon is Over," *https://www.linkedin.com/pulse/hadoop-honeymoon-over-martyn-jones*

[2] See "Low-Rank Matrix Factorizations at Scale: Spark for Scientific Data Analytics," Alex Gittens, MMDS 2016 in the C context, and "Size of Datasets for Analytics and Implications for R," Szilard Pafka, useR! 2016, in the R context.

[3] https://wiki.apache.org/hadoop/PoweredBy

In addition, this approach helps deal with the fact that Big Data sets may not fit into the memory of a single machine. (This aspect is too often overlooked in discussions of parallel/distributed systems.)

1.1 An Alternative to Hadoop/Spark

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 POR framework, no Java or other external language needed.
>
> The **partools** package is designed to meet these goals. It is intended as **a simple, sensible POR alternative to Hadoop/Spark**. Though not necessarily appropriate for all settings, for many R programmers, **partools** may be a much better choice than **Hadoop/Spark**.

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

1.2 Software Alchemy

Consider a large linear regression application. We might consider dividing the data into chunks, calling **lm()** on each chunk, then simply average the estimated coeficient vector over chunks. It can be shown that under fairly general conditions, this works, in the sense of producing estimators with the same asymptotic variance as that of the original estimator.

This technique was developed independently by a number of researchers. I call it Software Alchemy (SA), as it turns non-embarassingly parallel problems into embarassingly parallel ones.[4]

The **partools** package contains a number of functions using SA.

2 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 stems from just two simple sources:

(a) The package follows a Keep It Distributed philosophy: Form distributed objects *and keep using them in distributed form throughout one's R session, accessing them repeatedly for one's various desired operations — and avoiding, to the extent possible, operations that collect data from the worker nodes to the manager node.* By keeping things distributed in this way, whatever cost there was to distributing the data originally eventually yields a net performance gain.

(b) The package consists of a number of utility functions that greatly facilitate creating and storing and distributed objects, both in memory and on disk, and distributed applications such as for regression and classification.

[4] Software Alchemy: Turning Complex Statistical Computations into Embarrassingly-Parallel Ones, Norman Matloff, *Journal of Statistical Software*, 71 (2016).

3 Overview of the partools Package

The package is based on the following very simple principles, involving *distributed files* and *distributed data frames/matrices*. We'll refer to nondistributed files and data frames/matrices as *monolithic*.

- Files are stored in a distributed manner, in files with a common basename. For example, the virtual 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.

3.1 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:

3.1.1 Distributed-file and distributed-data frame functions

- **filesplit():** Create a distributed file from monolithic one.
- **filesplitrand():** Create a distributed file from monotlithic one, but randomize the record order.
- **filecat():** Create a monotlithic 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():** Returns the full name of the file chunk, associated with the calling cluster node, including suffix, e.g. '01', '02' etc.
- **distribsplit():** Create a distributed data frame/matrix from monotlithic one.
- **distribcat():** Create a monotlithic data frame/matrix from distributed one.

3.1.2 Tabulative functions

- **distribagg():** Distributed form of R's **aggregate()**.
- **distribcounts():** Wrapper for **distribagg()** to obtain table cell counts.
- **dfileagg():** Like **distribagg()**, but file-based rather than in-memory, in order to handle files that are too big to fit in memory, even on a distributed basis.
- **distribgetrows()**: Applies an R **subset()** or similar filtering operation to the distributed object, and collects the resulting rows into a single object at the caller.
- **distribrange():** Distributed form of R's **range()**.

3.2 Classical Computational Functions

- **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()**.

3.3 Modern Statistical/Machine Learning Functions

- **caclassfit():** Wrapper to do distributed fitting of a multiclass classification method such as random forests (**rpart** package) or SVM (**e1071** package).
- **caclasspred** Does prediction on new data from the output of **caclassfit()**.

3.4 Support Functions

- **formrowchunks():** Form chunks of rows of a data frame/matrix.
- **matrixtolist():** Forms 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.
- **dbs(), dbsmsg(), etc.:** Debugging aids.

3.5 More on Software Alchemy

All the functions with names beginning with "ca" use the Software Alchemy (SA) method. 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 one.[5]

A variant, suitable in many regression and classification algorithms, retains the chunk output in the result, rather than performing an avereging process. Consider for instance the use of LASSO for regression.[6] Each chunk may settle on a different subset of the predictor variables, so it would not make sense to average the estimated coefficient vectors. Instead, the predictor subsets and corresponding coefficients are retained in the SA output. When one is faced with predicting the

[5] In the world of parallel computation, the standard word for nonparallel is *serial*.

[6] Not currently available in the package, but easily coded under its framework.

response variable for a new data point, a prediction is calculated from each chunk, and those predictions are averaged to obtain the final prediction for the new point. In a classification context, voting may be used.

Note that SA requires that the data be i.i.d. If the data was stored in some sorted order — in the flight data below, it was sorted by date — it needs to be randomize it first, using one of the functions provided by **partools** for this purpose.

4 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 R virtual cluster in the sense of the R **parallel** package (which is loaded by **partools**). 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, in this case with $n = 16$. Indeed, one is often saddened to find that the parallel version is actually *slower* than the serial one!

To create our cluster **cls**, we ran

```
> cls <- makeCluster(16)  # from 'parallel' library
> setclsinfo(cls)  # from 'partools'
```

4.1 Distribute the Data

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 data 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. As noted earlier, random arrangement of the rows is required for SA.

The argument 2 here means that the suffixes are 2 digits, specifically '01', '02' and so on. So, we create files **yr2008r.01** etc. using **filesplit()**. The call to **fileread()** specifies that cluster node 1 will read the file **yr2008r.01**, cluster node 2 will read **yr2008r.02** and so on. Each node will place the data it reads into a data frame **yr2008**.

4.2 Distributed Aggregation of Summary Statistics

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

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

(In practice, though, this would not be done, i.e. we would not have *both* distributed *and* monotlithic versions of the data at the same time.)

The first operation run involved the package's distributed version of R's **aggregate()**. Here we wanted 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
```

What **distribagg()** did here was have each cluster node run **aggregate()** on its own chunk of the data, then (pardon the pun) aggregate the return values from the nodes.

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!

4.3 Un-distributing Data

Remember, the Keep It Distributed 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,])')
```

What **distribgetrows()** does is produce a data frame at each cluster node, per the user's instructions, then combine them together at the caller via R's **rbind()**. The user sets the second argument, a quoted string, to whatever she would have done on a serial basis. A simple concept, yet quite versatile.

4.4 Distributed NA Processing

As another example, say we are investigating data completeness. We may wish to flag all records having an inordinate number of NA values. As a first step, we may wish to add a column to our data frame, indicating how many NA values there 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 accord with the **partools** Keep It Distributed 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 our case here, 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.

4.5 SA: Linear Regression

Now let's turn to statistical operations, starting of course with linear regression. As noted, some **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 different from the non-distributed version, say in the third significant digit, it is just as accurate statistically; neither estimate is "better" than the other.

The SA function names begin with 'ca', for "chunk averaging." The SA version of **lm()**, for instance, is **calm()**.

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')$tht))
 (Intercept)     DepDelay     Distance
-1.061262941  1.019150592 -0.001213252
   user  system elapsed
 13.414   0.691  18.396
```

Note again the quoted-string argument. This is the one the user will give to **lm()** in the serial case.

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

4.6 SA: Principal ComponentsRegression

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
```

Once again, the second argument of **caprcomp()** is a quoted string, in which the user specifies the desired arguments to **prcomp()**. Since those arguments may be complicated, the code cannot deduce the number of variables, and thus needs to be specified in the third argument.

We have more than a 6-fold speedup here. Agreement of the component standard deviations is good.[7]

4.7 SA: K-Means Clustering

The package also includes a distributed version of k-means clustering. Here it is run 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
```

[7] Note that the last component, i.e. the eighth one, is minuscule, statistically 0. Again, SA gives results that are statistically equivalent to the serial ones.

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
```

4.8 SA: Multiclass Classification

Here we apply random forests to the UC Irvine forest cover type data, which has 590,000 records of 55 variables. The last variable is the cover type, coded 1 through 7. The machine used was quad core with hyperthreading level of 2, giving a theoretically possible parallelism degree of 8. Thus a cluster of size 8 was tried.

We used random forests for our classification algorithm (perhaps appropriate, given the data here!), but could have used any algorithm whose R implemention has a **predict()** method. (Some output not shown.)

```
> library(rpart)
> cls <- makeCluster(8)
> setclsinfo(cls)
> clusterEvalQ(cls,library(rpart))  # have each node load pkg
> cvr <- read.csv('~/Research/DataSets/ForestCover/CovTypeFull.csv',
     header=FALSE)
> cvr$V55 <- as.factor(cvr$V55)  # rpart requires factor for class
> trnidxs <- sample(1:580000,290000)  # form training and test sets
> trn <- cvr[trnidxs,]
```

```
> tst <- cvr[-trnidxs,]
> distribsplit(cls,'trn')
# do fit at each node; again, one argument is the serial argument
> system.time(fit <- caclassfit(cls,'rpart(V55 ~ .,data=trn,xval=25)'))
   user  system elapsed
  0.126   0.005  25.294
> system.time(predout <- caclasspred(fit,tst,55,type='class'))
   user  system elapsed
 26.132   0.910  27.045
> predout$acc
[1] 0.6692542
> system.time(fitser <- rpart(V55 ~ .,data=trn,xval=25))
   user  system elapsed
 87.699   0.143  87.850
> # system.time(predoutser <- predict(fitser,tst[,-10],type='class'))
> system.time(predoutser <- predict(fitser,tst[,-55],type='class'))
   user  system elapsed
  0.302   0.101   0.403
> mean(predoutser == tst[,55])
[1] 0.6691202
```

What **caclassfit()** does is run our classification algorithm, in this case **rpart**, at each node, then collect all the fitted models and return them to the caller. So here, after the call, **fit** will be an R list, element i of which is the fit computed by cluster node i.

The call to **caclasspred()** then applies these fits (on the parent node, not the cluster nodes) to our test data, **tst**. The result, **predout** contains the predictions for each of the 149 cases in **tst**, using voting among the eight fits.

Here SA more than tripled the fitting speed, with the same accuracy. Prediction using SA is relatively slow, due to the voting process, but arguably this is not an issue in a production setting, and in any case, SA was substantially faster even in total fitting plus prediction time.

5 Dealing with Memory Limitations

The discussion so far has had two implicit assumptions:

- The number of file chunks and the number of (R **parallel**) cluster nodes are equal, and the latter is equal to the number of physical computing devices one has, e.g. the number of cores in a multicore machine or the number of network nodes in a physical clsuter.
- Each file chunk fits into the memory[8] of the corresponding cluster node.

The first assumption is not very important. If for some reason we have created a distributed file with more chunks than our number of physical computing devices, we can still set up an R **parallel** cluster with size equal to the number of file chunks. Then more than one R process will run on at least some of the cluster nodes, albeit possibly at the expense of, say, an increase in virtual memory swap operations.

[8] Say, physical memory plus swap space.

The second assumption is the more pressing one. For this reason, the **partools** package includes functions such as **dfileagg()**. The latter acts similarly to **distribagg()**, but with a key difference: Any given cluster node will read from many chunks of the distributed file, and will process those chunks one at a time, never exceeding memory constraints.

Consider again our flight delay data set. As a very simple example, say we have a two-node physical cluster, and that each node has memory enough for only 1/4 of the data. So, we break up the original data file to 4 pieces, **yr2008.1** through **yr2008.4**, and we run, say,

```
# say we have machines pc28 and pc29 available for computation
> cls <- makeCluster(c('pc28','pc29'))
> dfileagg(cls,c('yr2008.1','yr2008.2','yr2008.3','yr2008.4'),
     c("DepDelay","ArrDelay","AirTime"),
     c("Origin","Dest"),"yr2008", FUN="max")
```

Our first cluster node will read **yr2008.1** and **yr2008.2**, one at a time, while the second will read **yr2008.3** and **yr2008.4**, again one at a time, At each node, at any given time only 1/4 of the data is in memory, so we don't exceed memory capacity. But they will get us the right answer, and will do so in parallel, roughly with a speedup factor of 2.

More functions like this will be added to **partools**.