

Big Data and Distributed Computing in R

Kylie Ariel Bemis

11/19/2018

Motivating example

What if flights was very large? E.g., multiple years, or more airports?

```
library(modelr)
library(nycflights13)

lmFit <- function(data) {
  lm(arr_delay ~ dep_delay, data=data)
}

fit1 <- lmFit(flights)
fit1
```

```
##
## Call:
## lm(formula = arr_delay ~ dep_delay, data = data)
##
## Coefficients:
## (Intercept)    dep_delay
##      -5.899         1.019
```

How can we fit this in parallel?

The easiest problems to parallelize are **embarrassingly parallel**:

- ▶ Independent tasks requiring no communication between them
- ▶ Data can be split into similarly-sized, independent subsets
- ▶ Almost anything that can be done with `lapply()` or `purrr::map()`

So how can we parallelize fitting a linear model to a single, large `flights` dataset?

- ▶ Fit a separate model for each year?
- ▶ Fit a separate model for each airport?
- ▶ Fit separate models to random samples (bootstrap) or random subsets (D&R)?

Divide and Recombine (D&R)

Consider a linear regression model with n observations:

$$Y = X\beta + \epsilon$$

Divide the the data into r subsets of m observations each, so $n = rm$.

For each subset s , the least-squares estimate is:

$$\dot{\beta}_s = (X_s' X_s)^{-1} X_s' Y_s$$

The D&R estimate for the coefficients β is then:

$$\ddot{\beta} = \frac{1}{r} \sum_{s=1}^r \dot{\beta}_s$$

Source: http://ml.stat.purdue.edu/hafen/preprints/Guha_Stat_2012.pdf

Partition the data into subsets

First, we partition the data into subsets.

```
n <- 10
set.seed(1)
partitions <- rep(1/n, n)
names(partitions) <- seq_len(n)

flights2 <- resample_partition(flights, partitions)
```

We also define a function for obtaining the D&R coefficients from the models.

```
DnR_coef <- function(x) {
  rowSums(sapply(x, coef) / length(x))
}
```

Fit the data on the models (single-threaded)

```
fitn <- lapply(flights2, lmFit)
```

```
coef(fit1)
```

```
## (Intercept)    dep_delay  
##    -5.899493      1.019093
```

```
DnR_coef(fitn)
```

```
## (Intercept)    dep_delay  
##    -5.899540      1.019098
```

The parallel package

The `parallel` package integrates two approaches to parallel and distributed computing in R.

- ▶ Multicore forking creates clones of the original R session
- ▶ Multiprocess SOCKET clusters create new R sessions

```
library(parallel)  
detectCores()
```

```
## [1] 4
```

Multicore forking approach

- ▶ Clones the original R session to create worker processes
- ▶ Requires OS forking support, only works on macOS and Linux
- ▶ Only works on a single machine, not over a network cluster
- ▶ Minimal overhead in communication between processes
- ▶ Worker processes share memory with the original R session
 - ▶ In theory, this is memory-efficient, as the data is shared and does not need to be exported or duplicated from the host R session
 - ▶ In practice, forked memory management is OS-dependent, and R's garbage collection frequently triggers duplication of objects

Using forking

`mclapply()` provides a multicore version of `lapply()` that uses forking.

```
fitmc <- mclapply(flights2, lmFit)

coef(fit1)
```

```
## (Intercept)    dep_delay
##    -5.899493      1.019093
```

```
DnR_coef(fitmc)
```

```
## (Intercept)    dep_delay
##    -5.899540      1.019098
```

You can create a forked cluster manually with `makeForkCluster()`.

Multiprocess SOCKET cluster approach

- ▶ Create new worker R sessions manually
- ▶ Relies on built-in R networking utilities, works on all OS's
- ▶ Works over a network cluster
- ▶ Communication between processes depends on network
- ▶ Worker processes are independent, fresh R sessions
 - ▶ Each process has an independent, separate memory space
 - ▶ Data must be moved manually to the worker processes
 - ▶ Libraries must be reloaded

Using sockets

Creating a SOCKET cluster works on all operating systems.

```
cl <- makeCluster(detectCores())  
result <- clusterEvalQ(cl, library(modelr))  
fitcl <- parLapply(cl, flights2, lmFit)  
stopCluster(cl)  
  
coef(fit1)
```

```
## (Intercept)    dep_delay  
##    -5.899493      1.019093
```

```
DnR_coef(fitcl)
```

```
## (Intercept)    dep_delay  
##    -5.899540      1.019098
```

Linear regression on a larger dataset

Suppose we want to run linear regression on a slightly larger dataset with more predictors.

```
n <- 1e6
p <- 9
b <- runif(p)
names(b) <- paste0("x", 1:p)
data <- vector("list", length=p + 1)
data[[p + 1]] <- rnorm(n)
for ( i in 1:p ) {
  xi <- rnorm(n)
  data[[i]] <- xi
  data[[p + 1]] <- data[[p + 1]] + xi * b[i]
}
data <- as.data.frame(data)
names(data) <- c(names(b), "y")
library(pryr)
object_size(data)
```

Linear regression on a larger dataset (cont'd)

To see how much memory it takes to run linear regression on this dataset, let's create a `mem_overhead` function.

```
mem_max <- function() {  
  pryr:::show_bytes(sum(gc()[,6] * c(pryr:::node_size(), 8)))  
}  
  
mem_overhead <- function(code) {  
  gc(reset=TRUE)  
  expr <- substitute(code)  
  eval(expr, parent.frame())  
  rm(code, expr)  
  finish <- mem_used()  
  pryr:::show_bytes(mem_max() - finish)  
}
```

Linear regression on large dataset (cont'd)

```
mem_overhead(fit2 <- lm(y ~ ., data=data))  
coef(fit2)
```

This typically uses several times the memory as the original dataset.

We can see that fitting a linear model with `lm()` on a large dataset uses quite a lot of memory overhead. Imagine if this dataset were gigabytes large instead of only 80 MB.

Partition the dataset for D&R

```
library(modelr)
n <- 10
partitions <- rep(1/n, n)
names(partitions) <- seq_len(n)

data2 <- resample_partition(data, partitions)
```

D&R regression using lapply

```
lmFit2 <- function(data) {  
  lm(y ~ ., data=data)  
}  
mem_overhead(fit2dr <- lapply(data2, lmFit2))  
coef(fit2)  
DnR_coef(fit2dr)
```


D&R regression using parLapply

```
cl <- makePSOCKcluster(detectCores())  
result <- clusterEvalQ(cl, library(modelr))  
fit2cl <- parLapply(cl, data2, lmFit2)  
stopCluster(cl)  
DnR_coef(fit2cl)
```

It takes significant overhead to export the data to each additional R session, especially if the data is very large.

On a single machine, fitting the model in parallel this way ends up taking *more* memory, because the data is loaded in multiple R sessions.

It would be great if we could share the same dataset between R sessions in a portable and reliable way.

Get indices of the subsets

We want to use a data structure that we can share between R sessions without having to copy the whole object. That means our `resample` objects from `resample_partition()` won't work anymore. However, we can still use the same subsets by extracting the row indices.

```
subidx <- sapply(data2, function(rs) rs$idx)  
str(subidx)
```

Packages for on-disk data

In addition to traditional RDBMS's, there are a growing number of packages dedicated to working with large datasets on-disk in R.

- ▶ `bigmemory` on CRAN
 - ▶ Uses shared memory to allow sharing of in-memory objects between R sessions
 - ▶ Also supports file-backed matrices in flat files
 - ▶ Extended by `bigalgebra` and `biganalytics` packages
- ▶ `ff` on CRAN
 - ▶ Uses on-disk flat files for data storage
 - ▶ Extended by `ffbase`
- ▶ `HDF5Array` on Bioconductor
 - ▶ Uses HDF5 files for data storage
 - ▶ Supports delayed block processing
- ▶ `matter` on Bioconductor
 - ▶ Uses on-disk files for data storage
 - ▶ Supports custom formats and some delayed operations

Using ff

With **ff**, we can use the `as.ffdf()` function to coerce our dataset into an **ff** data frame. This writes the data to disk as an efficiently-accessed binary flat file.

```
library(ff)
data.ff <- as.ffdf(data)
data.ff
```

D&R regression with ff

Because **ff** only needs to know the filename and its other metadata (such as dim and dimnames) are stored in R, it can be serialized to other R sessions easily.

```
cl <- makePSOCKcluster(detectCores())
clusterExport(cl, "data.ff")
result <- clusterEvalQ(cl, library(ff))
fit2ff <- parLapply(cl, subidx, function(i) {
  subdata <- as.data.frame(data.ff[i,])
  lm(y ~ ., data=subdata)
})
stopCluster(cl)
DnR_coef(fit2ff)
```

Considerations for out-of-memory data management

Can you use a RDBMS?

- ▶ Databases like MySQL and SQLite excel at managing data on disk
- ▶ Use independent file connections to access a database in parallel
- ▶ What operations are supported?
 - ▶ Often optimized for data transformation such joins and summarization
 - ▶ Not designed for linear algebra

When using a binary flat-file-based (or HDF5-based) approach:

- ▶ What types of data are allowed? (Strings are often forbidden)
- ▶ How is the data stored?
 - ▶ Matrices are typically stored in “column-major” order in R, and packages like `bigmemory` and `ff` use the same approach on-disk
 - ▶ Accessing data in one direction (columns) may be faster than accessing in another direction (rows)
- ▶ What operations are supported?
 - ▶ Often optimized for fast, random access rather than data manipulation
 - ▶ Some delayed operations may be supported (`matter` and `HDF5Array`)
 - ▶ Linear algebra may be supported

Considerations for parallel memory management

There is often a trade-off between CPU efficiency and memory efficiency.

For example, consider k-fold cross-validation:

- ▶ This problem is easily parallelized, as the model for each fold is trained independently of the others
- ▶ But parallel cross-validation may be very memory inefficient, as each fold actually uses the whole dataset
- ▶ Parallel k-fold cross-validation is potentially up to k-times faster, but may be memory-bound in R
- ▶ Sharing the same copy of the data between R sessions only really helps if the algorithm has low memory overhead
- ▶ For example `lm()` is very memory-inefficient, but `biglm()` from the `biglm` package tries to use less memory

Always consider how much speed you may gain by parallelizing code versus the total memory that may be used in a parallel implementation.

Try to use file-based formats (e.g., databases) that can be shared between R sessions in parallel when possible.

MapReduce

MapReduce is a programming model for parallel, distributed processing on big data.

Its name derives from the `Map()` and `Reduce()` functions used in functional programming languages such as R and LISP.

MapReduce was developed by Google as a specialization “**split-apply-combine**” strategy for data analysis.

The most well-known and widely-used implementation of MapReduce is Hadoop, but MapReduce is a general programming strategy that can be implemented in any language.

Split-Apply-Combine

Split-apply-combine, as described by Hadley Wickam:

Many data analysis problems involve the application of a split-apply-combine strategy, where you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together.

– Wickam 2011

Source:

<https://www.jstatsoft.org/index.php/jss/article/view/v040i01/v40i01.pdf>

Split-Apply-Combine (cont'd)

You have already been using split-apply-combine in your data analyses whenever you use dplyr!

Consider the following code that computes the average distance flown to each destination in the `flights` dataset:

```
library(dplyr)
flights %>%
  group_by(dest) %>%
  summarise(mean(distance, na.rm=TRUE))
```

First, we **split** the dataset using `group_by()`.

Next, we **apply** the function `mean()` to each group.

Finally, we **combine** the results in `summarise()`.

MapReduce (cont'd)

At a high level, MapReduce **splits** the input data into subsets.

Then a Map() function is **applied** to each subset, and outputs key-value pairs.

A shuffle step organizes the output results by their keys.

Then a Reduce() function is applied to **combine** the values for each key.

The final results are then collated and returned.

Map() and Reduce()

The name derives from the Map() and Reduce() functions commonly found in functional programming languages such as R and LISP.

```
x <- list(1:10, 11:20, 21:30)
y <- list(1, 2, 3)
```

```
map.out <- Map("+", x, y) # add elements of 'x' and 'y'
map.out
```

```
## [[1]]
##  [1]  2  3  4  5  6  7  8  9 10 11
##
## [[2]]
##  [1] 13 14 15 16 17 18 19 20 21 22
##
## [[3]]
##  [1] 24 25 26 27 28 29 30 31 32 33
```

```
reduce.out <- Reduce("+", map.out) # sum each element
reduce.out
```

```
##  [1] 39 42 45 48 51 54 57 60 63 66
```

MapReduce visualized

Because MapReduce is typically performed in a distributed manner, there is no way of knowing what part of the data will be processed on what node, so MapReduce passes data as key-value pairs, and uses the keys to organize where intermediate results need to go.

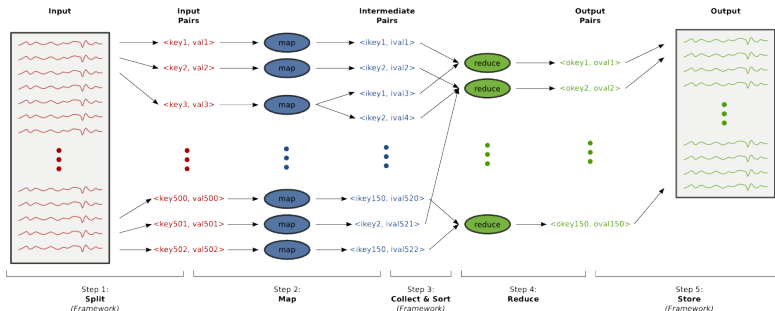


Figure 1: <http://webmapreduce.sourceforge.net>

Map step

In the Map step, the data is split into subsets.

The data is then passed to a `Map()` function as key-value pairs.

The `Map()` function processes each key-value pair and outputs a list of new key-value pairs.

`Map(k1,v1) → list(k2,v2)`

Shuffle step

In the Shuffle step, the output of the Map step is sorted by key.

All of the values with the same key are grouped together.

These intermediate key-value pairs are then passed to the Reduce step.

```
Shuffle(list(list(k2,v2))) → list(k2,list(v2))
```

Reduce step

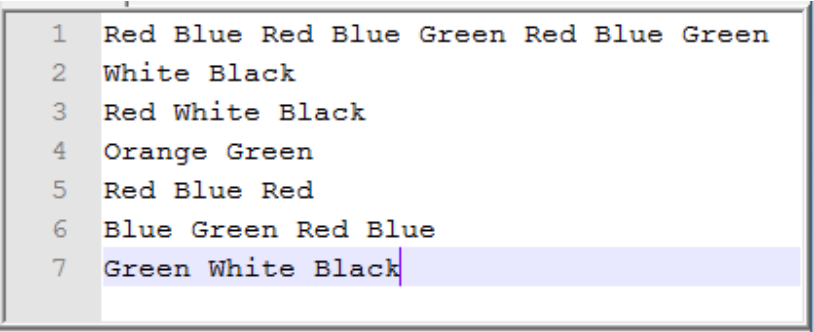
In the Reduce step, the intermediate key-value pairs are passed to the `Reduce()` function.

The `Reduce()` function processes the values for the input key, and outputs a final output value for that same key

`Reduce(k2, list (v2)) → list(v3)`

Word count example

Suppose we want to count the number of each word in a document.

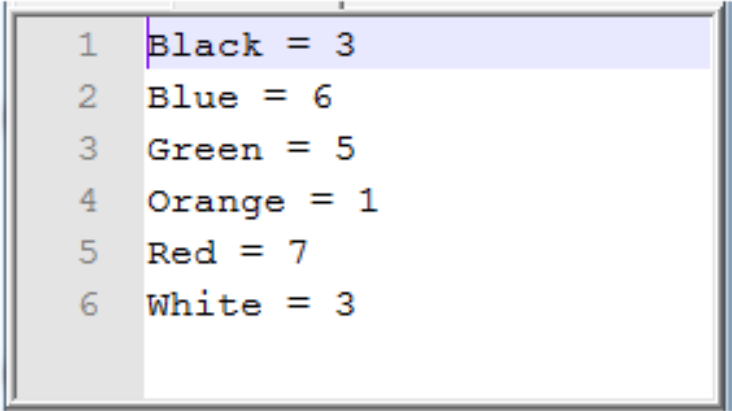


```
1 Red Blue Red Blue Green Red Blue Green
2 White Black
3 Red White Black
4 Orange Green
5 Red Blue Red
6 Blue Green Red Blue
7 Green White Black
```

Figure 2: <https://www.journaldev.com/8848/mapreduce-algorithm-example>

Word count example (cont'd)

This is the final output we would expect. How does MapReduce get us here?

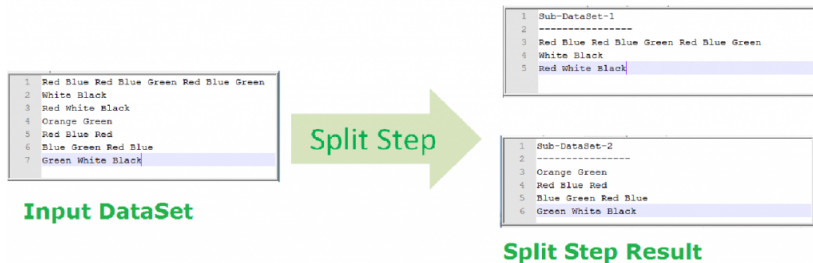


```
1 Black = 3
2 Blue = 6
3 Green = 5
4 Orange = 1
5 Red = 7
6 White = 3
```

Final Output

Word count example: Map step (split)

First, the data is split into subsets.



MapReduce - Split Step

Figure 4: <https://www.journaldev.com/8848/mapreduce-algorithm-example>

Word count example: Map step (Map function)

The Map() function outputs keys (words) and values (1). Later, the Reduce() function will sum these values up to get the count for each word.

```
1 Sub-DataSet-1
2 -----
3 Red Blue Red Blue Green Red Blue Green
4 White Black
5 Red White Black
```

```
1 Sub-DataSet-2
2 -----
3 Orange Green
4 Red Blue Red
5 Blue Green Red Blue
6 Green White Black
```

Split Step Result

Mapping Step

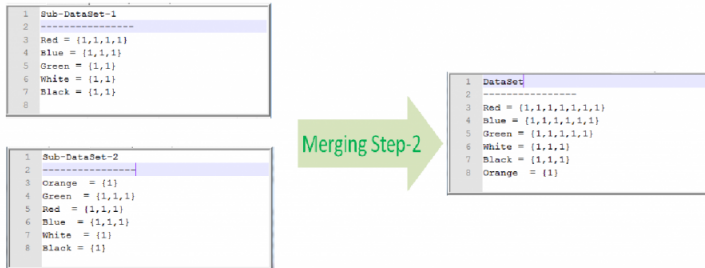
```
1 Sub-DataSet-1
2 -----
3 Red = 1
4 Blue = 1
5 Red = 1
6 Blue = 1
7 Green = 1
8 Red = 1
9 Blue = 1
10 Green = 1
11 White = 1
12 Black = 1
13 Red = 1
14 White = 1
15 Black = 1
```

```
1 Sub-DataSet-2
2 -----
3 Orange = 1
4 Green = 1
5 Red = 1
6 Blue = 1
7 Red = 1
8 Blue = 1
9 Green = 1
10 Red = 1
11 Blue = 1
12 Green = 1
13 White = 1
14 Black = 1
```

MapReduce - Mapping Step

Word count example: Shuffle step (merge)

The output of each Map() function are merged together.



MapReduce - Merging Step 2

Figure 6: <https://www.journaldev.com/8848/mapreduce-algorithm-example>

Word count example: Shuffle step (sort)

The merged output from the Map() functions are sorted by key.



MapReduce - Sorting Step

Figure 7: <https://www.journaldev.com/8848/mapreduce-algorithm-example>

Word count example: Reduce step (Reduce function)

The Reduce() function sums the values for each key to get the word count.



MapReduce - Reduce Step

Figure 8: <https://www.journaldev.com/8848/mapreduce-algorithm-example>