Memory- and Computation-Efficient Statistical Tools for Big Matrices

R-Lille (April 2022)

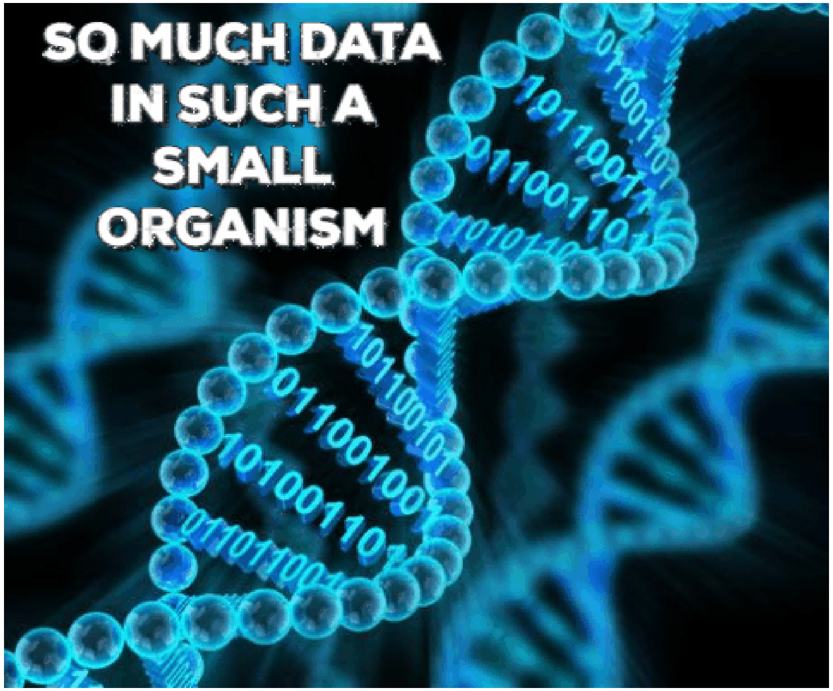
Florian Privé

Senior Researcher, Aarhus University (DK)

Motivation

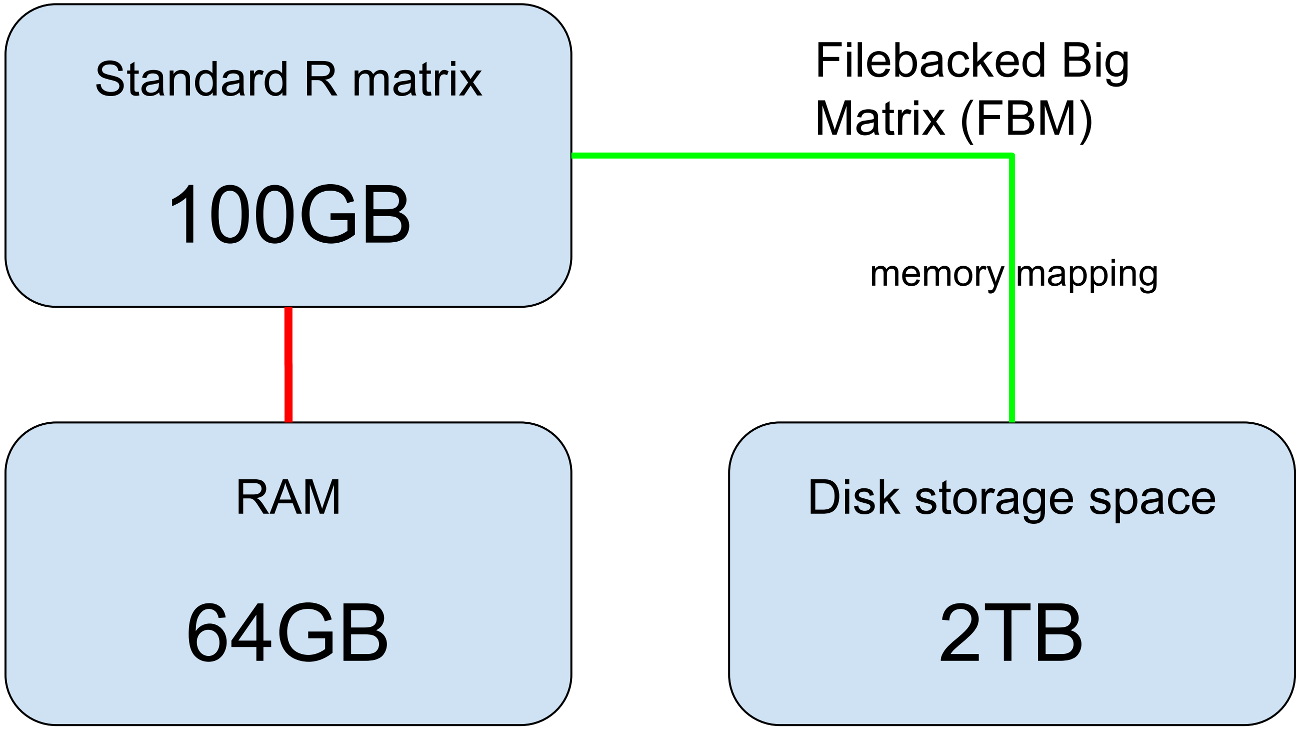
Working with very large genotype matrices

- previously: 15K x 280K, celiac disease (~30GB)
- currently: 500K x 500K, UK Biobank (~2TB)



But I still want to use R..

The solution I found



Format `FBM` is very similar to format `filebacked.big.matrix` from package {bigmemory} (details in this vignette).

How memory-mapping works

- when you access the 1st element (1st row, 1st col), it accesses a block (say the first column) from disk into memory (RAM)
- when you access the 2nd element (2nd row, 1st col), it is already in memory so it is accessed very fast
- when you access the second column, you access from disk again (once)
- you can access many columns like that, until you do not have enough memory anymore
- some space is freed automatically so that new columns can be accessed into memory

Everything is seamlessly managed by the operating system (OS).

Simple accessors

Similar accessor as R matrices

```r
X <- FBM(2, 5, init = 1:10, backingfile = "test")
```

```r
X$backingfile
```

```
## [1] "C:\\Users\\au639593\\Desktop\\R-presentation\\test.bk"
```

```r
X[, 1]   ## ok
```

```
## [1] 1 2
```

```r
X[1, ]   ## bad
```

```
## [1] 1 3 5 7 9
```

```r
X[]      ## super bad
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

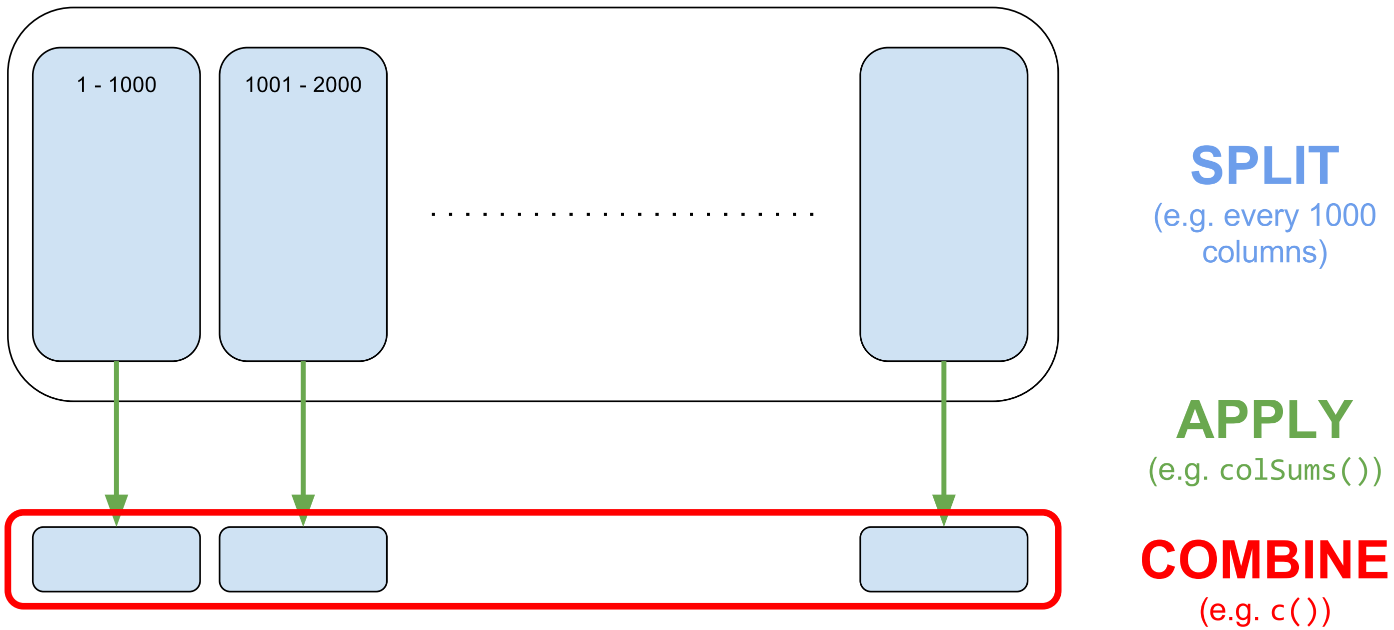
Similar accessor as R matrices

```r
colSums(X[])  ## super bad
```

```
## [1]  3  7 11 15 19
```

Split-(par)Apply-Combine Strategy

Apply standard R functions to big matrices (in parallel)



Implemented in `big_apply()`.

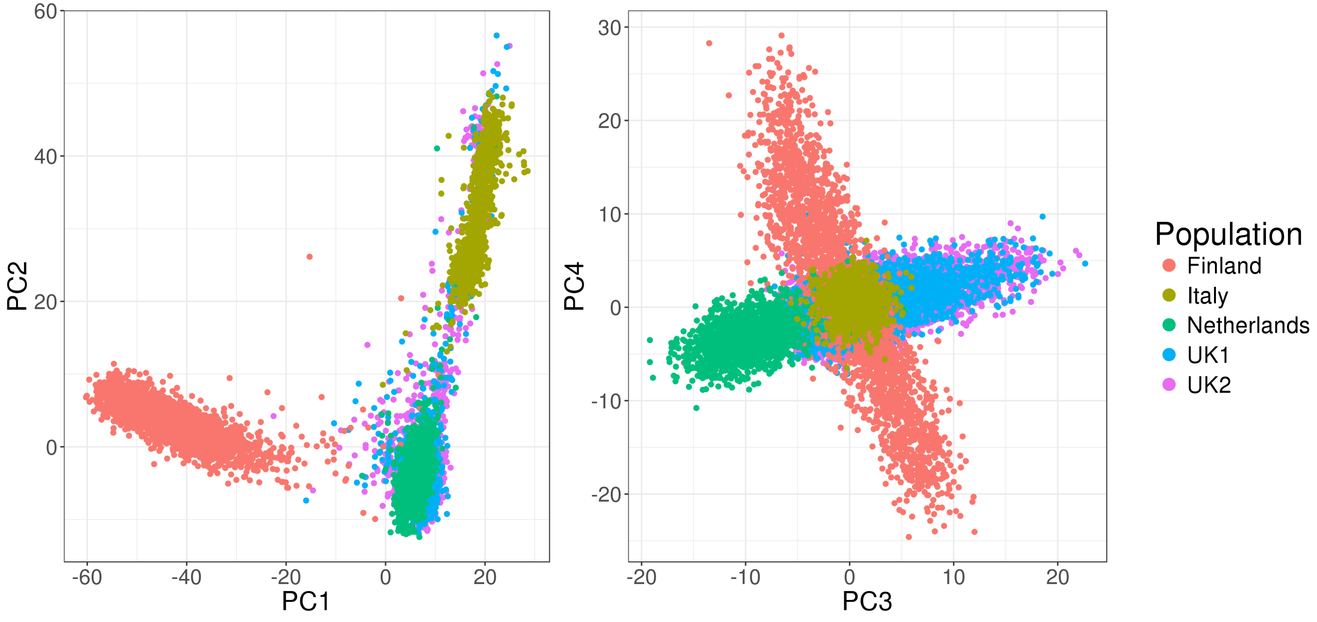
Similar accessor as Rcpp matrices

```
// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::depends(bigstatsr, rmio)]]
#include <bigstatsr/BMAcc.h>

// [[Rcpp::export]]
NumericVector big_colsums(Environment BM) {

  XPtr<FBM> xpBM = BM["address"]; // get the external pointer
  BMAcc<double> macc(xpBM);       // create an accessor to the data

  size_t n = macc.nrow();
  size_t m = macc.ncol();

  NumericVector res(m);

  for (size_t j = 0; j < m; j++)
    for (size_t i = 0; i < n; i++)
      res[j] += macc(i, j);

  return res;
}
```

Some examples from my work

Partial Singular Value Decomposition

15K $\times$ 100K -- 10 first PCs -- 6 cores -- **1 min** (vs 2h in base R)



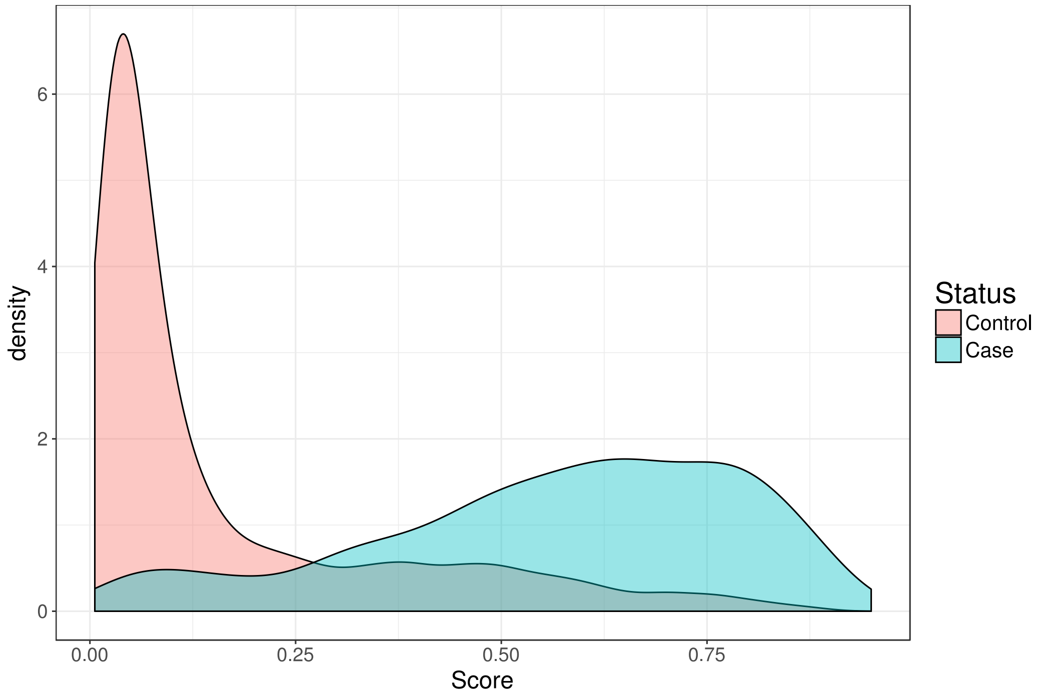
Implemented in `big_randomSVD()`, powered by R packages {RSpectra} and {Rcpp}.

Sparse linear models

Predicting complex diseases with a penalized logistic regression

15K $\times$ 280K -- 6 cores -- **2 min** (10x faster than {glmnet})

Automatic (parallel) grid-search for the two hyper-parameters of elastic-net.



Let us try some functions

from R package {bigstatsr}

Create an FBM object

```r
X <- FBM(10e3, 1000, backingfile = "test2")$save()
object.size(X)
```

```
## 680 bytes
```

```r
file.size(X$backingfile)  ## 8 x 1e4 x 1e3
```

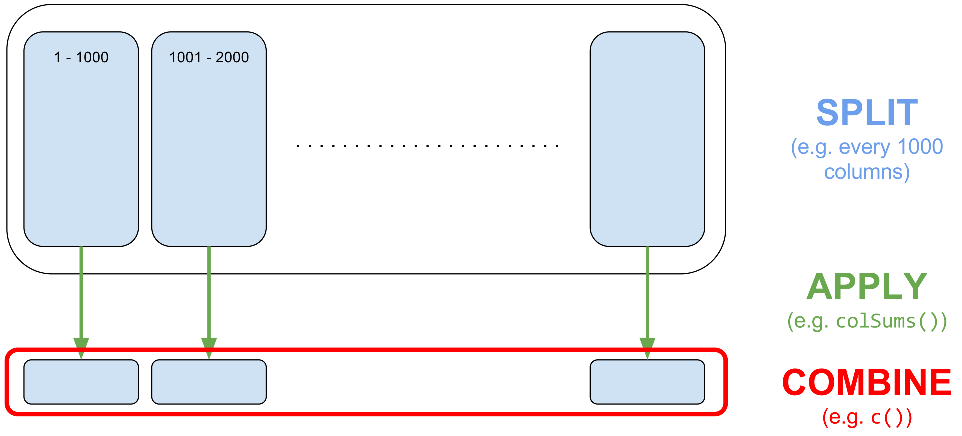
```
## [1] 8e+07
```

```r
typeof(X)
```

```
## [1] "double"
```

```r
# `$save()` stored the object in an .rds file
# which you can reload in any R session
X <- big_attach("test2.rds")
```

Fill the FBM with random values



```
big_apply(X, a.FUN = function(X, ind) {
  X[, ind] <- rnorm(nrow(X) * length(ind))
  NULL  ## Here, you don't want to return anything
}, a.combine = 'c')
```

```
## NULL
```

```
X[1:5, 1]
```

```
## [1] -0.63074508  1.55563752 -0.02059371 -0.13974420 -0.73635154
```

Correlation matrix

```
mat <- X[]
system.time(corr1 <- cor(mat))
```

```
##    user  system elapsed
##    6.11    0.00    6.15
```

```
system.time(corr2 <- big_cor(X))
```

```
##    user  system elapsed
##    5.98    0.04    6.04
```

```
all.equal(corr1, corr2[])
```

```
## [1] TRUE
```

Partial Singular Value Decomposition

```
system.time(svd1 <- svd(scale(mat), nu = 10, nv = 10))
```

```
##    user  system elapsed 
##   24.53    0.09   25.06
```

```
# Quadratic in the smallest dimension, linear in the other one
system.time(svd2 <- big_SVD(X, fun.scaling = big_scale(), k = 10))
```

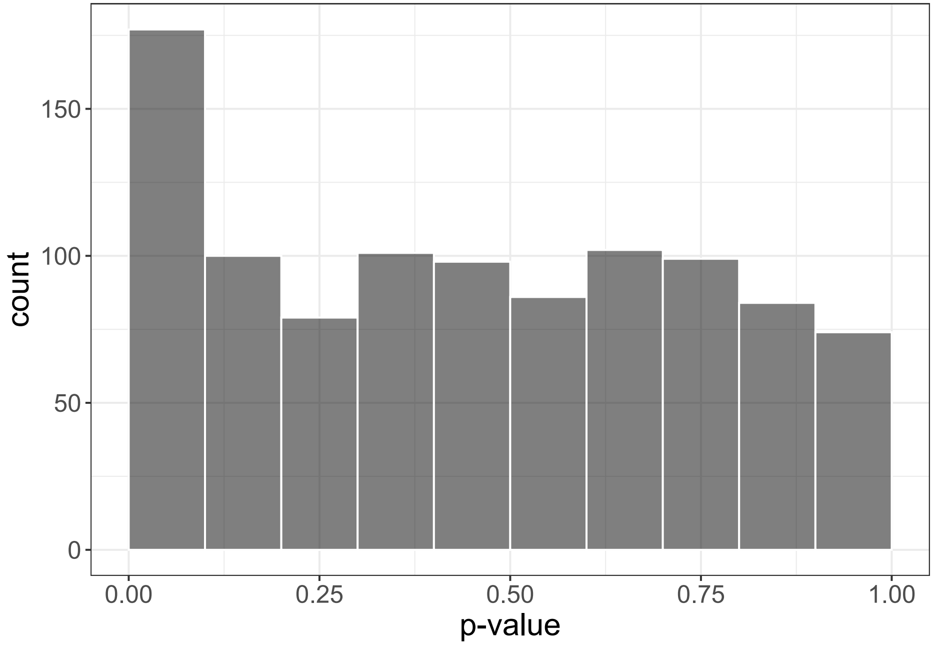
```
##    user  system elapsed 
##    6.98    0.07    7.14
```

```
# Linear in both dimensions
# Extremely useful if both dimensions are very large
system.time(svd3 <- big_randomSVD(X, fun.scaling = big_scale(), k =
```

```
##    user  system elapsed 
##    2.20    0.00    2.21
```

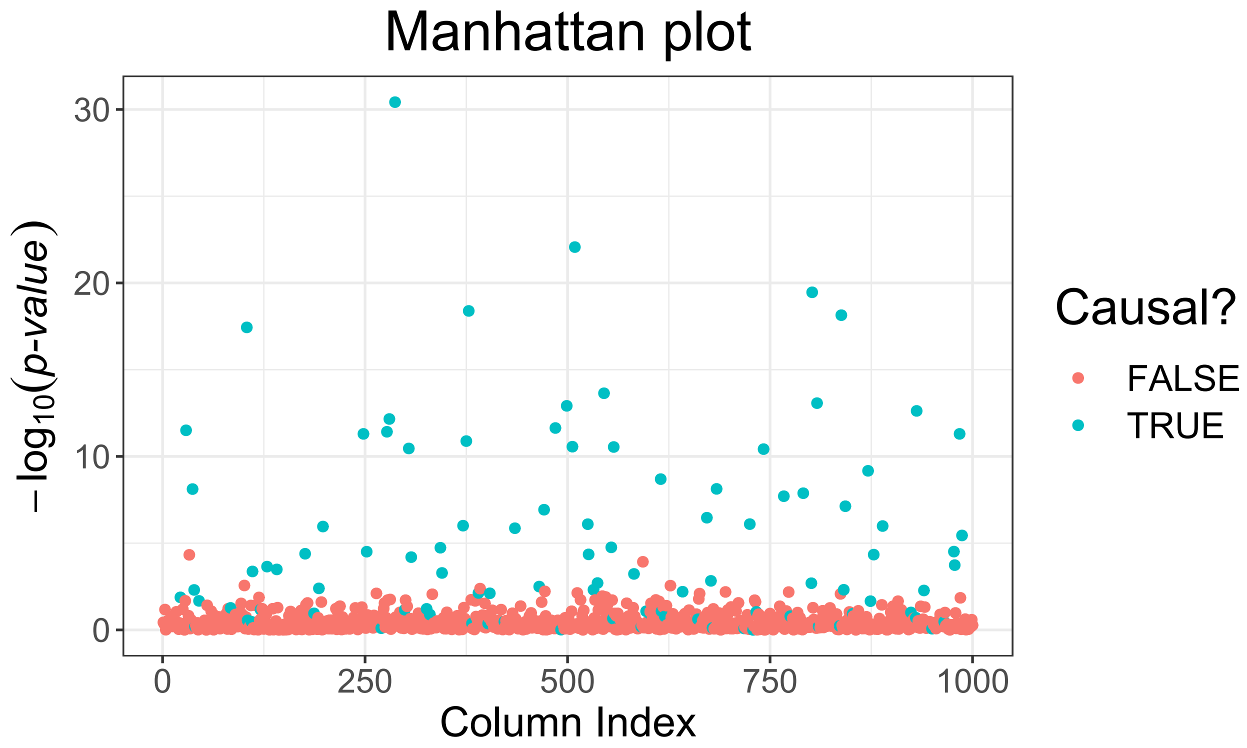
Multiple association

```r
M <- 100 # number of causal variables
set <- sample(ncol(X), M)
y <- scale(X[, set]) %*% rnorm(M)
y <- y + rnorm(length(y), sd = 2 * sd(y))

mult_test <- big_univLinReg(X, y, covar.train = svd2$u)
plot(mult_test)
```



Multiple association

```r
library(ggplot2)
plot(mult_test, type = "Manhattan") +
  aes(color = cols_along(X) %in% set) +
  labs(color = "Causal?")
```



Prediction

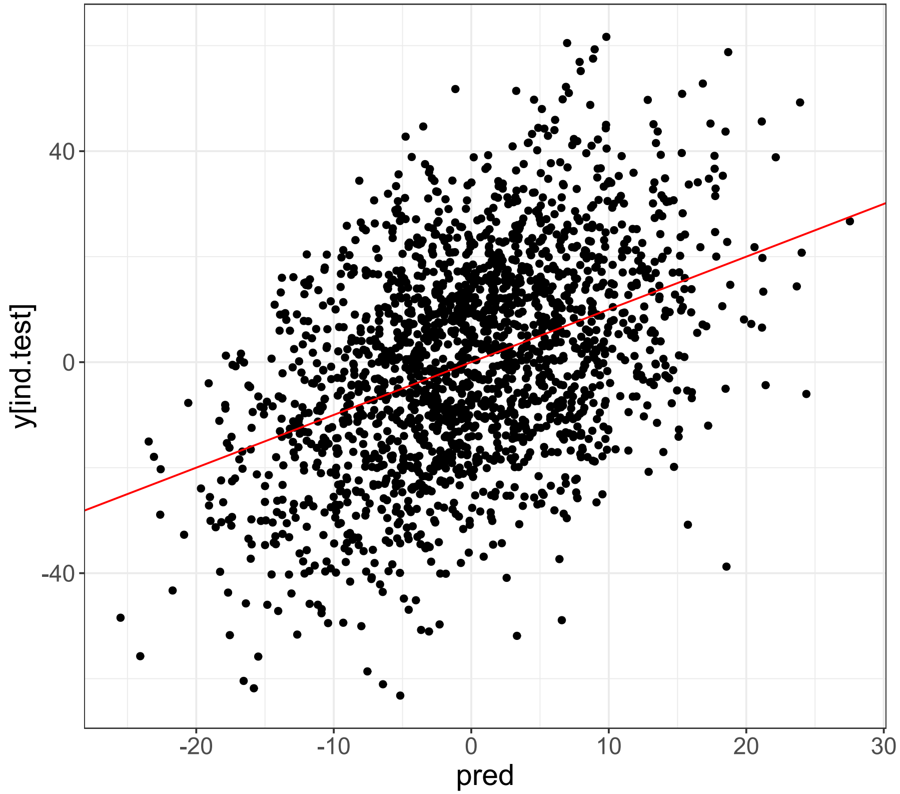
```r
# Split the indices in train/test sets
ind.train <- sort(sample(nrow(X), size = 0.8 * nrow(X)))
ind.test <- setdiff(rows_along(X), ind.train)

# Train a linear model with elastic-net regularization
# and automatic choice of hyper-parameter lambda
train <- big_spLinReg(
  X, y[ind.train],
  ind.train = ind.train,                # use a subset for training
  covar.train = svd2$u[ind.train, ], # use additional covariables
  pf.covar = rep(0, ncol(svd2$u)),    # do not penalize covariables
  alphas = c(1, 0.1, 0.01))           # try a grid of values for alpha
```

```r
# Get predictions for the test set
pred <- predict(train, X = X, ind.row = ind.test,
                covar.row = svd2$u[ind.test, ])
```

A tutorial on fitting penalized regressions in this vignette.

Prediction

```r
# Plot true value vs prediction
qplot(pred, y[ind.test]) +
  geom_abline(intercept = 0, slope = 1, color = "red") +
  theme_bigstatsr()
```



Toy case:

Compute the sum for each column

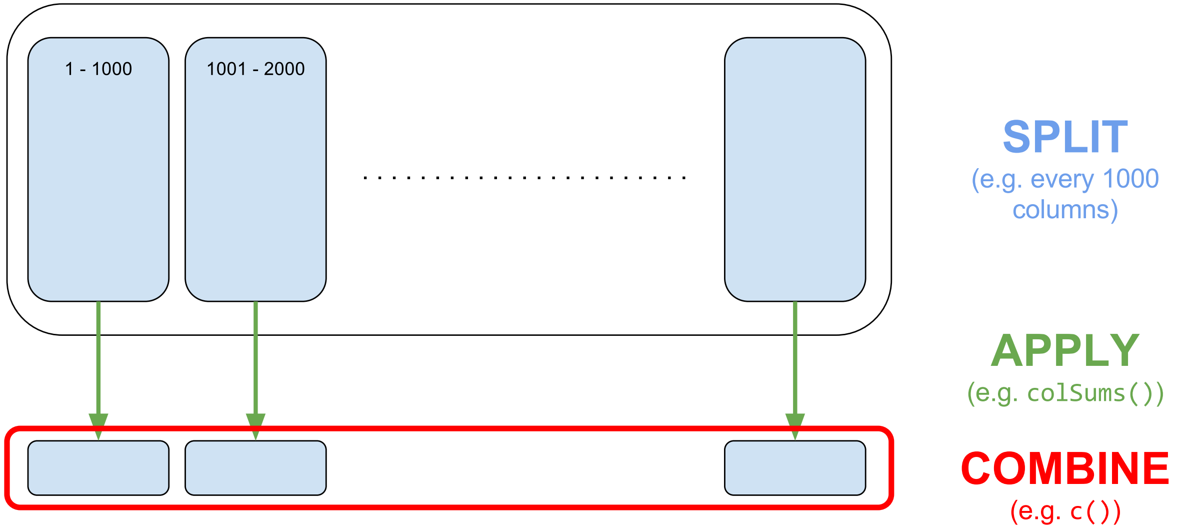
Brute force solution

```r
sums1 <- colSums(X[])  ## /!\ access all the data in memory
```

Do it by blocks

```r
sums2 <- big_apply(X, a.FUN = function(X, ind) colSums(X[, ind]),
                   a.combine = 'c')

all.equal(sums2, sums1)
```

```
## [1] TRUE
```



Using Rcpp (1/3)

```cpp
// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::depends(bigstatsr, rmio)]]
#include <bigstatsr/BMAcc.h>

// [[Rcpp::export]]
NumericVector bigcolsums(Environment BM) {

  XPtr<FBM> xpBM = BM["address"]; // get the external pointer
  BMAcc<double> macc(xpBM);       // create an accessor to the data

  size_t n = macc.nrow(), m = macc.ncol();
  NumericVector res(m);  // vector of m zeros

  for (size_t j = 0; j < m; j++)
    for (size_t i = 0; i < n; i++)
      res[j] += macc(i, j);

  return res;
}
```

Using Rcpp (1/3)

```r
sums3 <- bigcolsums(X)

all.equal(sums3, sums1)
```

```
## [1] TRUE
```

Using Rcpp (2/3): the bigstatsr way

```
// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::depends(bigstatsr, rmio)]]
#include <bigstatsr/BMAcc.h>

// [[Rcpp::export]]
NumericVector bigcolsums2(Environment BM,
                          const IntegerVector& rowInd,
                          const IntegerVector& colInd) {

  XPtr<FBM> xpBM = BM["address"];
  SubBMAcc<double> macc(xpBM, rowInd, colInd, 1);

  size_t n = macc.nrow(), m = macc.ncol();
  NumericVector res(m);  // vector of m zeros

  for (size_t j = 0; j < m; j++)
    for (size_t i = 0; i < n; i++)
      res[j] += macc(i, j);

  return res;
}
```

Using Rcpp (2/3): the bigstatsr way

```r
sums4 <- bigcolsums2(X, rows_along(mat), cols_along(mat))
all.equal(sums4, sums1)
```

```
## [1] TRUE
```

```r
sums5 <- bigcolsums2(X, rows_along(mat), 1:10)
all.equal(sums5, sums1[1:10])
```

```
## [1] TRUE
```

Using Rcpp (3/3): already implemented

```
sums6 <- big_colstats(X)
str(sums6)
```

```
## 'data.frame':    1000 obs. of  2 variables:
##  $ sum: num  -30.5 152.1 148 -102.1 44.6 ...
##  $ var: num  1.03 1 1.01 1 0.99 ...
```

```
all.equal(sums6$sum, sums1)
```

```
## [1] TRUE
```

Parallelism

Most of the functions are parallelized

```r
ind.rep <- rep(cols_along(X), each = 100)  ## size: 100,000
(NCORES <- nb_cores())
```

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

```r
system.time(
  mult_test2 <- big_univLinReg(X, y, covar.train = svd2$u,
                               ind.col = ind.rep)
)
```

```
##    user  system elapsed
##    9.14    0.00    9.28
```

```r
system.time(
  mult_test3 <- big_univLinReg(X, y, covar.train = svd2$u,
                               ind.col = ind.rep, ncores = NCORES)
)
```

```
##    user  system elapsed
##   18.70    0.03    5.09
```

Parallelize your own functions

```
system.time(
  mult_test4 <- big_parallelize(
    X, p.FUN = function(X, ind, y, covar) {
      bigstatsr::big_univLinReg(X, y, covar.train = covar,
                                ind.col = ind)
    }, p.combine = "rbind", ind = ind.rep,
    ncores = NCORES, y = y, covar = svd2$u)
)
```

```
##    user  system elapsed
##    0.06    0.04    7.12
```

```
all.equal(mult_test4, mult_test3)
```

```
## [1] TRUE
```

Conclusion

I'm able to run algorithms

on 100GB of data

in R on my computer

Advantages of using FBM objects

- you can apply algorithms on **data larger than your RAM**,
- you can easily **parallelize** your algorithms because the data on disk is shared,
- you write **more efficient algorithms** (you do less copies and think more about what you're doing),
- you can use **different types of data**, for example, in my field, I’m storing my data with only 1 byte per element (rather than 8 bytes for a standard R matrix). See the documentation of the FBM class for details.

Two other packages

{bigsparser}

Provides a Sparse matrix format with data on disk with some features:

- convert from a dgCMatrix/dsCMatrix to an SFBM, a Sparse Filebacked Big Matrix
- compute the product and crossproduct of an SFBM with a vector
- solve Ax=b, where A is a symmetric SFBM and b is a vector
- a new *compact* format is available, which is useful when non-zero values in columns are contiguous (or almost).

This package is intended for more efficient use of sparse data in C++ and also when parallelizing, since data on disk does not need copying.

{bigsnpr}

Extends {bigstatsr} (and also uses {bigsparser}) with functions specific to genetic SNP data:

- to convert between formats, especially to an FBM
- wrappers around PLINK (e.g. for quality controls)
- special functions for PCA/SVD
- polygenic scores (predictors based on genetic data) methods
- multiple testing
- many utility functions and other algorithms
- functions that work directly on memory-mapped PLINK bed/bim/fam files (often using the same code, just with a different accessor)

Contributions and extensions are welcome!

If we meet in-person someday, make sure to ask for an hex sticker

Thanks!

Presentation available at
https://privefl.github.io/R-presentation/bigrverse.html

— @privefl —

Slides created using R package **xaringan**