*Working with* R

A University of Queensland Advanced Workshop

# Session 12:

# An Introduction to Parallel Programming

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

# Contents

## 1 Why would anyone want to do that?

Under normal circumstances (for statisticians!) you probably don't.

- Modern computers have large memory (relative to the past) and mostly they also have multiple "cores"[a]. This allows them to do (in effect) several things at once. Sometimes you want to call upon several of them rather than just the one (usually) your process has been allocated.

- They can also have fast internet connections to other machines that be called upon to share a computational load, for a very large computation.

Parallel processing allows you to harness these additional resources to tackle typically very large jobs, perhaps needing days to finish. A fairly recent book, (?), offers some useful techniques and insights.

[a]Desktops and laptops will usually have 4, 8 or 16. "Graphical Processing Units" (GPU's) will typically have thousands.

## 1.1 *Embarrassingly* parallel problems

These problems are of a kind where

- the computation job can be subdivided into several sub-jobs, *all of the same kind*,
- which can be run independently of each other (i.e. *no inter-process communication*) and
- at the end, the results can be collected up *in any order*.

Restrictive as it appears, a surprising number of real problem fit the paradigm. For example, in Statistics, most simulation and bootstrapping problems qualify.

In more complex "process" models, the computational sub-jobs are typically much more varied and interconnected and more subtle and delicate programming is required.

We will only consider *embarrassingly parallel* problems here.

## 2 The core package: `parallel`

This package provides various low-level tools, as well as a suite of simple high-level tools for *embarrassingly parallel* applications which mirror the standard interlooping tools, such as *lapply*, *sapply*, &c, plus a few. A good quick way to get a start is to look at the help information for a key function, such as:

```
library(parallel)
?clusterApply
```

Working through a simple example makes the process clearer.

Recalling the *BirthWt* logistic regression, lets do a brute-force check that the optimal regressions we found using the stepwise procedure are, in fact, best in a very large field of possible models.

As this is a logistic regression problem, the models have to be fitted “from scratch”, one after another.

First, consider a non-parallel approach. We need to do a bit of preparatory work beforehand to establish the field of candidate models.

```
nam <- c(setdiff(names(BirthWt), "low"),
         "poly(age,2)", "smoke*ui", "age*ftv", "poly(lwt,2)")
g <- (2 + log(nrow(BirthWt)))/2  ## "Goldilocks" penalty

all_forms <- "low~1"
for(n in nam) {
  all_forms <- c(all_forms, paste(all_forms, n, sep = "+"))
}
all_forms <- sub("1\\+", "", all_forms)
all_forms[1:8]
```

```
[1] "low~1"              "low~age"            "low~lwt"
[4] "low~age+lwt"        "low~race"           "low~age+race"
[7] "low~lwt+race"       "low~age+lwt+race"
```

```
length(all_forms)
```

```
[1] 4096
```

We will fit all logistic regressions with the main effect terms, together with a few well-chosen curvilinear and interaction terms.

The sequential computation (single core)

```
t1 <- system.time({
  res <- vapply(all_forms, function(form) {
    thisCall <- substitute(glm(FORM, binomial, BirthWt),
                           list(FORM = as.formula(form)))
    thisModel <- eval(thisCall)
    thisLogL <- logLik(thisModel)
    c(AIC = AIC(thisModel),
      BIC = BIC(thisModel),
      GIC = AIC(thisModel, k = g),
      logL = as.vector(thisLogL),
      DF = as.integer(attr(thisLogL, "df")))
  }, FUN.VALUE = c(AIC = 0.0, BIC = 0.0, GIC = 0.0,
                   logL = 0.0, DF = 0L))
})
t1  ## how long did that take?

   user  system elapsed
  9.314   0.008   9.323
```

Note especially how the output is kept as small as possible, retaining all essential information.

Now the parallel version (on various machines; this is a unix machine with 12 cores):

```
cl <- makePSOCKcluster(max(1, detectCores()-1))      ## set up cores-1 machines
clusterEvalQ(cl, library(WWRData)) %>% invisible()
clusterExport(cl, "g")                               ## the all have "g"

t2 <- system.time({                                  ## farm out the jobs
  resP <- parSapply(cl, all_forms, function(form) {
    thisCall <- substitute(glm(FORM, binomial, BirthWt),
                           list(FORM = as.formula(form)))
    thisModel <- eval(thisCall)
    thisLogL <- logLik(thisModel)
    c(AIC = AIC(thisModel),
      BIC = BIC(thisModel),
      GIC = AIC(thisModel, k = g),
      logL = as.vector(thisLogL),
      DF = as.integer(attr(thisLogL, "df")))
  })
})
stopCluster(cl)
rm(cl)
```

Now for some checks. Did it work? How much faster was it?

```
all.equal(res, resP)

[1] TRUE

rbind(t1, t2)[, c("user.self", "sys.self", "elapsed")]

   user.self sys.self elapsed
t1     9.314    0.008   9.323
t2     0.009    0.000   1.914
```

Now we need to see which were the “best” models.

```r
results <- data.frame(formula = all_forms, t(res),
                      stringsAsFactors = FALSE) %>%
  arrange(logL, DF) %>%
  filter(!(c(1, diff(logL)) < 1e-6 & c(1, diff(DF)) == 0))

best10_AIC <- results %>% arrange(AIC) %>% filter(AIC < AIC[11])
best10_BIC <- results %>% arrange(BIC) %>% filter(BIC < BIC[11])
best10_GIC <- results %>% arrange(GIC) %>% filter(GIC < GIC[11])

list(AIC = as.formula(best10_AIC$formula[1]),
     BIC = as.formula(best10_BIC$formula[1]),
     GIC = as.formula(best10_GIC$formula[1]))
```

```
$AIC
low ~ lwt + smoke + ptl + ht + smoke * ui + age * ftv

$BIC
low ~ lwt + ptl + ht

$GIC
low ~ lwt + ptl + ht + ui + age * ftv
```

Now compare with what *step_AIC* would have produced, starting from our largest model:

```
BW0 <- glm(low ~ ., binomial, BirthWt)
scope <- list(lower = ~1, upper = ~.^2+poly(age, 2)+poly(lwt,2))
list(AIC = step_AIC(BW0, scope = scope),
     BIC = step_BIC(BW0, scope = scope),
     GIC = step_AIC(BW0, scope = scope, k = g)) %>%
       lapply(formula)

$AIC
low ~ age + lwt + smoke + ptl + ht + ui + ftv + age:ftv + smoke:ui


$BIC
low ~ lwt + ptl + ht


$GIC
low ~ lwt + race + smoke + ptl + ht
```

Close, but not the same!

# 3 The `doParallel` package approach

The *doParallel* package offers another approach to acccessing the parallel processing technology, which has gained wide acceptance in some areas of application, as it offers a bit more flexibility than the core *parallel* package on which it is built.[a]

## 3.1 Iterators

(See the vignettes in the `iterators` and `foreach` package for more.)

- A special type of **R** object that once set up, "feeds out" part of an object in sequence. They may be finite, or unending.
- Mainly used in conjunction with the *foreach* function (of the eponymous package.)

[a]Note that this is only one of many alternative parallel processing **R** packages we could have used, though.

Examples:

```
suppressPackageStartupMessages({
  library(doParallel)
})
search()

 [1] ".GlobalEnv"          "package:doParallel"   "package:iterators"
 [4] "package:foreach"     "package:parallel"     ".R_Cache"
 [7] ".Robjects"           "package:WWRCourse"    "package:WWRUtilities"
....
[28] "package:knitr"       "package:stats"        "package:graphics"
[31] "package:grDevices"   "package:utils"        "package:datasets"
[34] "package:methods"     "Autoloads"            "package:base"
```

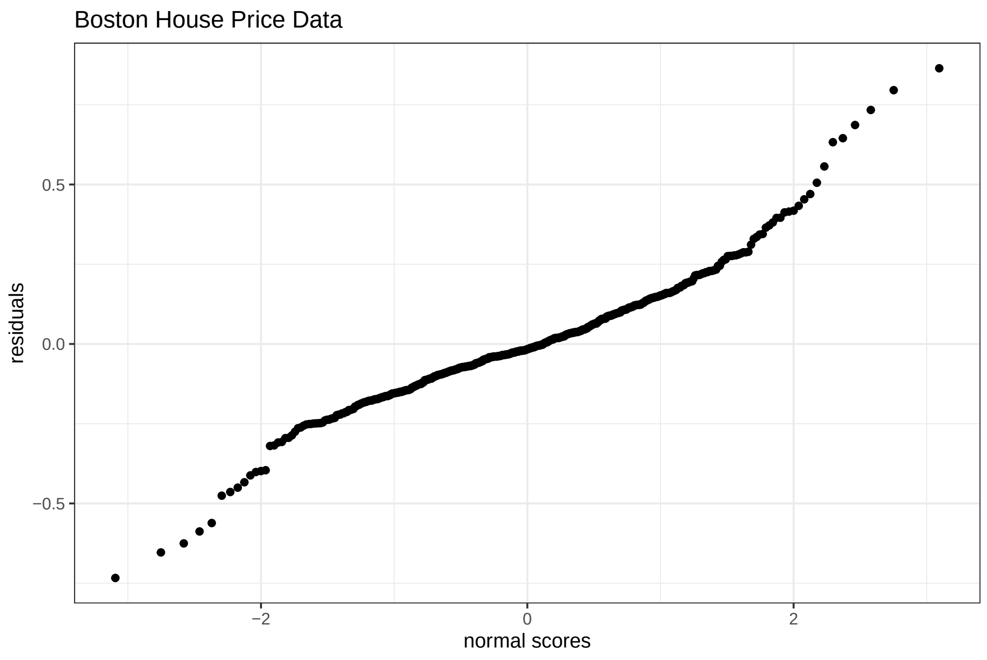
A slim example:

```
X <- matrix(1:20, 5, 4) %>% print

     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20

itXrows <- iter(X, by = "row")
Xt <- foreach(x = itXrows, .combine = cbind) %do% rev(x) ## NOT parallel yet!
Xt

     result.1 result.2 result.3 result.4 result.5
[1,]       16       17       18       19       20
[2,]       11       12       13       14       15
[3,]        6        7        8        9       10
[4,]        1        2        3        4        5
```

## 3.2 A simulation envelope for a normal scores plot

The Boston House Price data.

```r
hp_model <- lm(log(medv) ~ ., Boston)
rdat  <- data.frame(res = resid(hp_model))
ggplot(rdat) + aes(sample = res) + geom_qq() +
  labs(y = "residuals", x = "normal scores",
       title = "Boston House Price Data")
```



The strategy is

- Simulate a large number of observation vectors using only the fitted model itself (ie. not the observations directly)
- Re-fit the model again for each simulation to get simulated residuals
- Finally check where the actual residuals sits relative to the ideal simulated residual distributions.

To speed things up (hopefully!) we will do the simulations in parallel, spreading the work across several cores. We use the `doParallel` package, (Revolution Analytics and Weston, 2015), which also adds `foreach`, `iterators` and `parallel` itself to the search path.

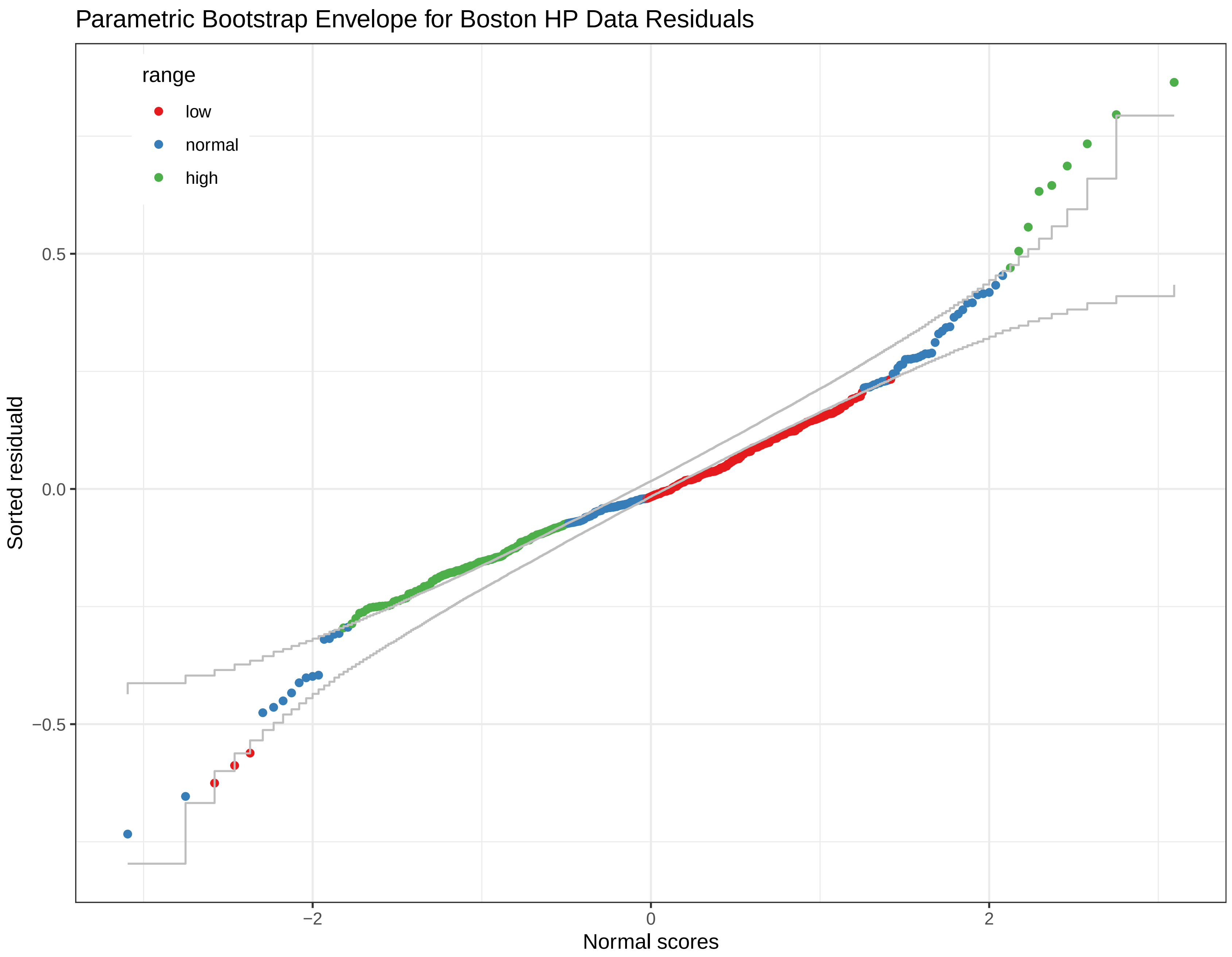
A fundamental tool:

```
simulate                    ## An S3 generic function in package:stats

function (object, nsim = 1, seed = NULL, ...)
UseMethod("simulate")
<bytecode: 0x55dd74702800>
<environment: namespace:stats>

methods("simulate")   ## for what kinds of model does it cater?

[1] simulate.lm*     simulate.negbin* simulate.nls*    simulate.polr*
see '?methods' for accessing help and source code

########################  initialisation
library(doParallel)
cores <- detectCores() %>% print

[1] 12

cl <- makeCluster(cores - 1)
registerDoParallel(cl)        ## all set to go
```

```r
N <- 10000                          ## Number of simulations, go big time
chunks <- idiv(N, chunks = cores - 1)  ## farm one bit out to each core?
QR <- hp_model$qr
Res <- foreach(nsim = chunks,      ## fed out by the iterator.
               .combine = cbind,
               .inorder = FALSE,  ## allows load balancing
               .export = "Boston") %dopar% {
                 Sims <- simulate(hp_model, nsim = nsim)
                 Sims <- qr.resid(QR, as.matrix(Sims))

                 ## line below equivalent to: apply(Sims, 2, sort)
                 matrix(Sims[order(col(Sims), Sims)], nrow = nrow(Sims))
               }
itRes <- iter(Res, by = "row")
Res <- foreach(r = itRes, .combine = rbind) %dopar%
  ## quantile(r, prob = c(0.025, 0.975))  ## too stringent?
  sort(r)[c(50, length(r)-49)]             ## just clip off the fuzz

##################### finalisation
stopCluster(cl)
rm(cl)
```

Now assemble the pieces and make a picture:

```r
colnames(Res) <- c("Low", "Upp")
rdat <- cbind(rdat, Res) %>%
  within({
    ns <- qnorm(ppoints(length(res)))  ## normal scores
    res <- sort(res)
    range <- ifelse(res <  Low, "low", ifelse(res > Upp, "high", "normal"))
    range <- factor(range, levels = c("low", "normal", "high"))
  })

### show the results
ggplot(rdat) + aes(x = ns, y = res, colour = range) + geom_point() +
  geom_step(aes(y = Low), direction = "hv", col = "grey") +
  geom_step(aes(y = Upp), direction = "vh", col = "grey") +
  labs(x="Normal scores", y="Sorted residuald",
  title="Parametric Bootstrap Envelope for Boston HP Data Residuals") +
  theme(legend.position = c(0.1, 0.9)) +
  scale_colour_brewer(palette = "Set1")
```


Parametric Bootstrap Envelope for Boston HP Data Residuals
range
low
normal
high
0.5
0.0
−0.5
Sorted residuald
−2
0
2
Normal scores

## 3.3 Combining random forests

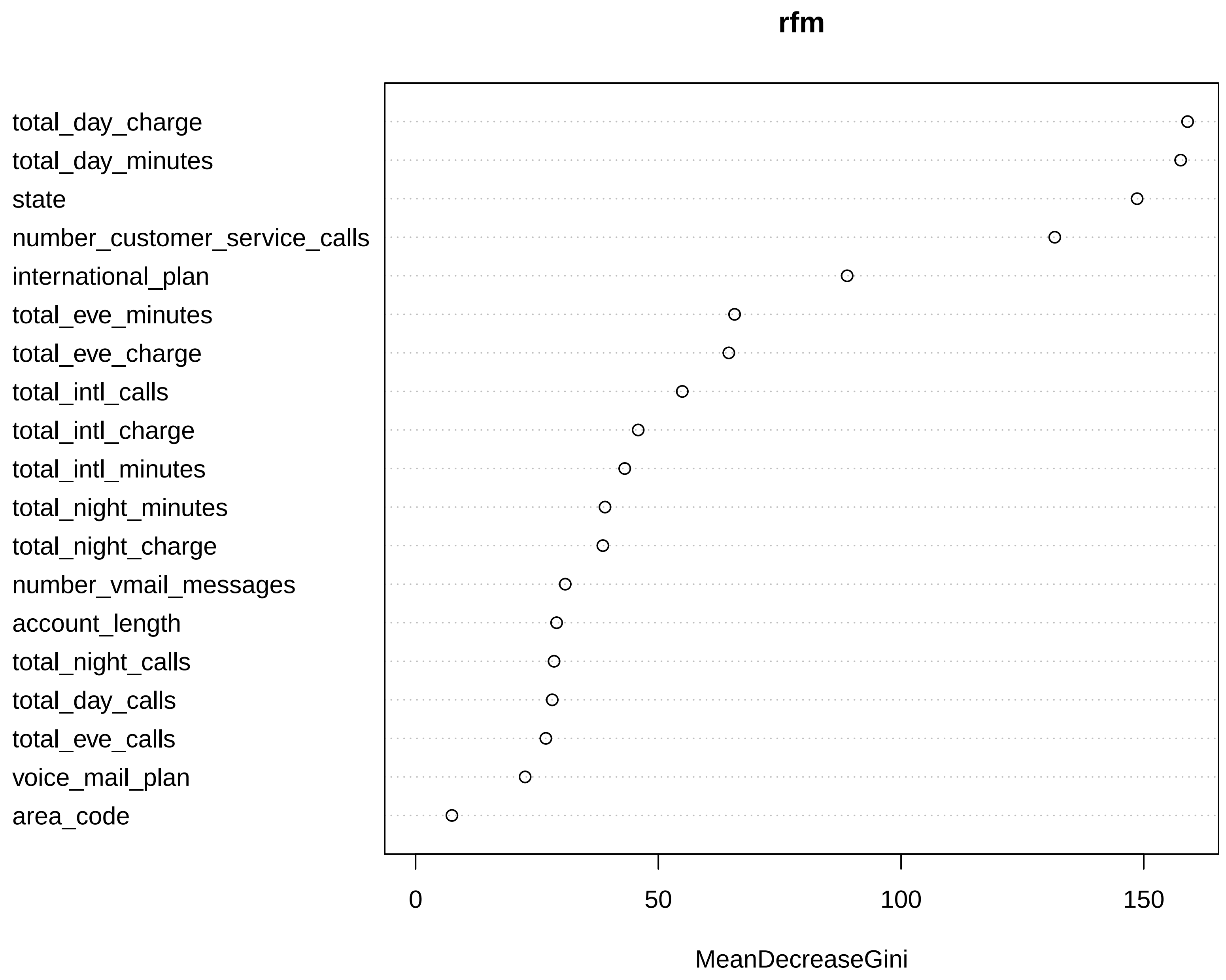
The `randomForest` fitting function is typically very fast, but for large data sets we might want to farm out some of the calculation to several cores, or several *nodes*. If we do so, how do we re-assemble the separate trees into one random forest object? The package itself has provided a function, *`combine`*, for just this purpose.

The churn "data" again.

```
fname <- system.file("extdata", "churnData.csv.gz", package = "WWRCourse")
churnData <- read_csv(gzfile(fname),     ## neater read (for eventual notebook)
                      col_types = cols(.default = col_double(),
                                       state = col_character(),
                                       area_code = col_character(),
                                       international_plan = col_character(),
                                       voice_mail_plan = col_character(),
                                       churn = col_character(),
                                       sample = col_character())) %>%
  unclass() %>% data.frame(stringsAsFactors = TRUE) %>% select(-sample)
names(churnData) %>% noquote()

 [1] state                          account_length
 [3] area_code                      international_plan
 [5] voice_mail_plan                number_vmail_messages
....
[15] total_night_charge             total_intl_minutes
[17] total_intl_calls               total_intl_charge
[19] number_customer_service_calls churn
```

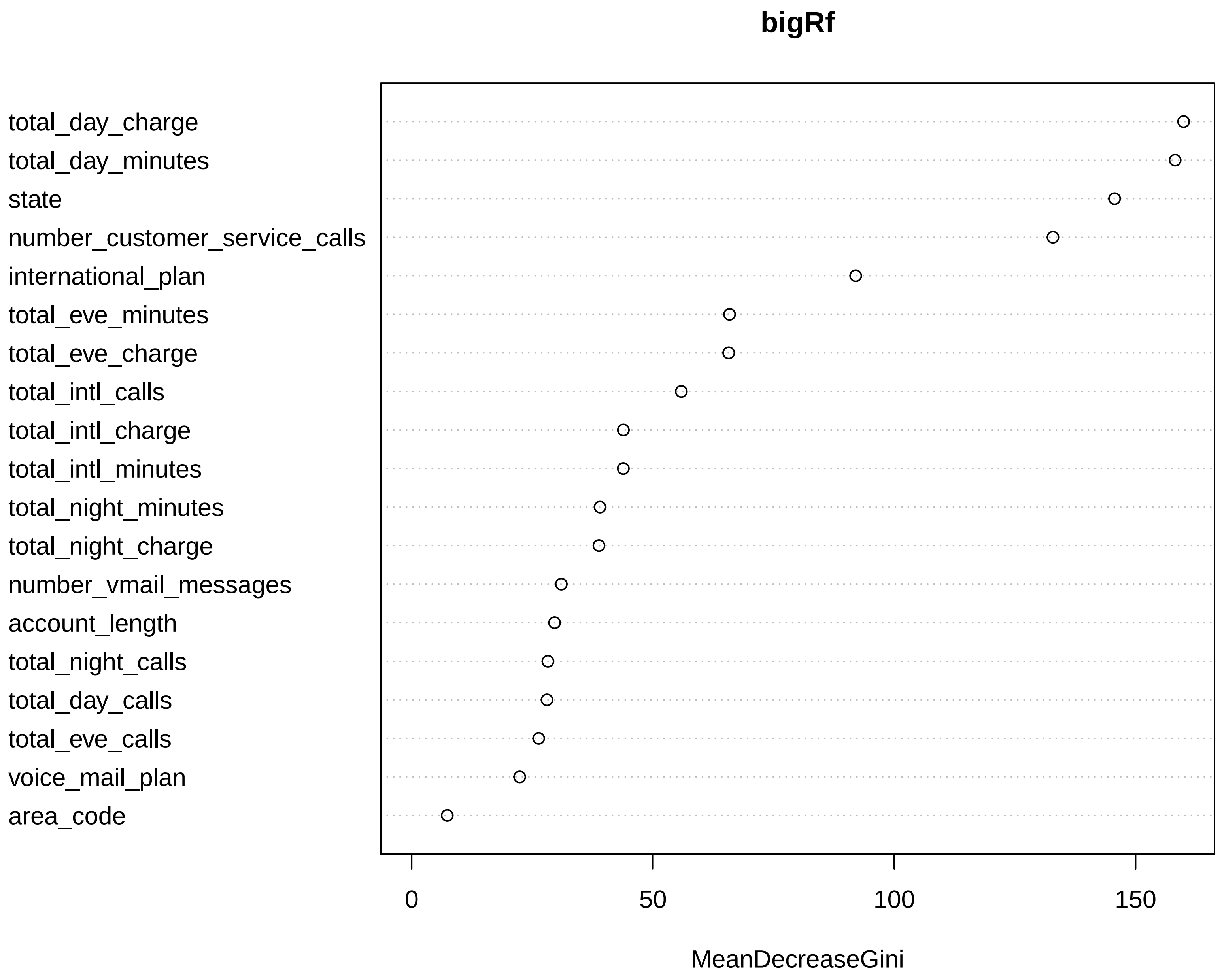
Consider a random forest approach.

```
suppressPackageStartupMessages(library(randomForest))
rfm <- randomForest(churn ~ ., churnData)
rfm


Call:
 randomForest(formula = churn ~ ., data = churnData)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 4.16%
Confusion matrix:
      no yes class.error
no  4242  51     0.01188
yes  157 550     0.22207

## note how redundant 'charge' and 'minutes'variables get level pegging
varImpPlot(rfm)
```


rfm
total_day_charge
total_day_minutes
state
number_customer_service_calls
international_plan
total_eve_minutes
total_eve_charge
total_intl_calls
total_intl_charge
total_intl_minutes
total_night_minutes
total_night_charge
number_vmail_messages
account_length
total_night_calls
total_day_calls
total_eve_calls
voice_mail_plan
area_code
0
50
100
150
MeanDecreaseGini

The randomForest package has tools that allow the forest to be built up progressively in parallel.

```r
cl <- makeCluster(cores - 2)
registerDoParallel(cl)          ## all set to go
ntree <- idiv(5000, chunks = max(5, cores - 2))
t1 <- system.time({
  bigRf <- foreach(ntree = ntree,
                   .combine = randomForest::combine, ## one in dplyr!
                   .packages = "randomForest",
                   .inorder = FALSE) %dopar% {
                     randomForest(churn ~ ., churnData, ntree = ntree)
                   }
})
stopCluster(cl)
rm(cl)
varImpPlot(bigRf)
```


bigRf
total_day_charge
total_day_minutes
state
number_customer_service_calls
international_plan
total_eve_minutes
total_eve_charge
total_intl_calls
total_intl_charge
total_intl_minutes
total_night_minutes
total_night_charge
number_vmail_messages
account_length
total_night_calls
total_day_calls
total_eve_calls
voice_mail_plan
area_code
0
50
100
150
MeanDecreaseGini

Now for a serial version:

```
ntree <- idiv(5000, chunks = max(5, cores - 2))
t2 <- system.time({
  bigRf <- foreach(ntree = ntree,
                   .combine = randomForest::combine, ## one in dplyr!
                   .packages = "randomForest",
                   .inorder = FALSE) %do% {       ### <<<--- only change!
                     randomForest(churn ~ ., churnData, ntree = ntree)
                   }
})
rbind(serial = t2, parallel = t1)[, c("user.self", "sys.self", "elapsed")]
```

```
         user.self sys.self elapsed
serial      24.672     0.38  25.053
parallel     2.212     0.36   7.427
```

## 4 Bootstrap aggregation revisited

We look again at the problem of using parallelism to speed up the construction of a forest of bootstrapped trees. (Cf. presentation 09).

Recall that the non-parallel version looked like this:

```r
bagRpart <- local({
  bsample <- function(dataFrame) # bootstrap sampling
    dataFrame[sample(nrow(dataFrame), rep = TRUE),  ]
  function(object, data = eval.parent(object$call$data),
           nBags=600, type = c("standard", "bayesian"), ...) {
    type <- match.arg(type)
    bagsFull <- vector("list", nBags)
    if(type == "standard") {
      for(j in 1:nBags)
          bagsFull[[j]] <- update(object, data = bsample(data))
      } else {
        nCases <- nrow(data)
        for(j in 1:nBags)
            bagsFull[[j]] <- update(object, data = data, weights = rexp(nCases))
        }
    class(bagsFull) <- "bagRpart"
    bagsFull
  }
})
```

For the parallel version, we need an iterator for random exponential deviates. This can be done using a trick.

```
library(doParallel) ## parallel backend; includes other pkgs
(cores <- detectCores())  ## how many real cores?

[1] 12

cl <- makeCluster(cores-1)
registerDoParallel(cl)
```

A modified version of the fitting function. Some care needed:

```
bagRpartParallel <- local({
  bsample <- function(dataFrame) # bootstrap sampling
    dataFrame[sample(nrow(dataFrame), rep = TRUE),  ]

  function(object, data = eval.parent(object$call$data),
           nBags = 600, type = c("standard", "bayesian"), ...,
           cores = detectCores() - 1, seed0 = as.numeric(Sys.Date())) {
    type <- match.arg(type)
    bagsFull <- foreach(j = idiv(nBags, chunks=cores), seed = seed0+seq(cores),
                        .combine = c, .packages = c("rpart", "stats"),
                        .inorder = FALSE, .export = c("bsample")) %dopar% {
                          ## now inside a single core
                          set.seed(seed = seed)
                          if(type == "standard") {
                            replicate(j, simplify = FALSE,
                                      update(object, data = bsample(data)))
                          } else {
                            replicate(j, simplify = FALSE,
                                      update(object, data = data,
                                             weights = rexp(nrow(data))))
                          }
                          ## end of single core mode
                        }
    class(bagsFull) <- "bagRpart"
    bagsFull
  }
})
```

## A timing comparison

```
set.seed(12345) ##
library(rpart)
Obj <- rpart(medv ~ ., Boston, minsplit = 5, cp = 0.005)  ## expand the tree
rbind(
  simple    = system.time(HPSBag  <- bagRpart(Obj, nBags=600)),
  bayes     = system.time(HPBBag  <- bagRpart(Obj, nBags=600, type="bayes")),
  parallel_s= system.time(HPSBagP <- bagRpartParallel(Obj, nBags=600)),
  parallel_b= system.time(HPBBagP <- bagRpartParallel(Obj, nBags=600,
                                                   type="bayes")))[,
                                    c("user.self", "sys.self", "elapsed")]

           user.self sys.self elapsed
simple         5.998    0.000   5.998
bayes          5.409    0.000   5.409
parallel_s     0.181    0.008   1.008
parallel_b     0.059    0.024   0.883

rm(Obj)
```

```
stopCluster(cl)  ## release resources
rm(cl)
```

Exercises:

- Check that the parallel version `barRpartParallel` still works on the credit card data.

- The `Boston` data has a continuous response whereas the credit card data had a categorical response. The predictor we wrote previously only catered for categorical responses.

  Write a general (enough) prediction method for `"bagRpart"` objects, i.e. general enough to handle both classification and regression tree cases.

One solution:

```r
predict.bagRpart <- function(object, newdata, ...,
                             method = object[[1]]$method) {
  switch(method,
         class = {
           X <- sapply(object, predict, newdata = newdata, type = "class")
           candidates <- levels(predict(object[[1]], type = "class"))
           X <- t(apply(X, 1, function(r) {
             table(factor(r, levels = candidates))
           }))
           factor(candidates[max.col(X)], levels = candidates)
         },
         anova = {
           X <- sapply(object, predict, newdata = newdata, type = "vector")
           rowMeans(X)
         },
         NA)
}
```

## References

Revolution Analytics and S. Weston (2015). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.10.

# Session information

**Date: 2021-01-29**

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 20.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils
- Other packages: doParallel 1.0.16, dplyr 1.0.3, english 1.2-5, forcats 0.5.1, foreach 1.5.1, GGally 2.1.0, ggplot2 3.3.3, ggthemes 4.2.4, gridExtra 2.3, haven 2.3.1, iterators 1.0.13, knitr 1.31, lattice 0.20-41, lme4 1.1-26, Matrix 1.3-2, mboost 2.9-4, mgcv 1.8-33, microbenchmark 1.4-7, nlme 3.1-151, patchwork 1.1.1, purrr 0.3.4, randomForest 4.6-14, rbenchmark 1.0.0, Rcpp 1.0.6, readr 1.4.0, rpart 4.1-15, scales 1.1.1, SOAR 0.99-11, stabs 0.6-3, stringr 1.4.0, tibble 3.0.5, tidyr 1.1.2, tidyverse 1.3.0, visreg 2.7.0, WWRCourse 0.2.3, WWRData 0.1.0, WWRGraphics 0.1.2, WWRUtilities 0.1.2, xtable 1.8-4
- Loaded via a namespace (and not attached): assertthat 0.2.1, backports 1.2.1, boot 1.3-26, broom 0.7.3, cellranger 1.1.0, cli 2.2.0, codetools 0.2-18, colorspace 2.0-0, compiler 4.0.3, crayon 1.3.4, DBI 1.1.1, dbplyr 2.0.0, digest 0.6.27, ellipsis 0.3.1, evaluate 0.14, fansi 0.4.2, farver 2.0.3, Formula 1.2-4, fractional 0.1.3, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.0.3, gtable 0.3.0, highr 0.8, hms 1.0.0, httr 1.4.2, inum 1.0-1, jsonlite 1.7.2, labeling 0.4.2, lazyData 1.1.0, libcoin 1.0-7, lifecycle 0.2.0, lubridate 1.7.9.2, magrittr 2.0.1, MASS 7.3-53, minqa 1.2.4, modelr 0.1.8, munsell 0.5.0, mvtnorm 1.1-1, nloptr 1.2.2.2, nnls 1.4, partykit 1.2-11, PBSmapping 2.73.0, pillar 1.4.7, pkgconfig 2.0.3, plyr 1.8.6, quadprog 1.5-8, R6 2.5.0, RColorBrewer 1.1-2, readxl 1.3.1, reprex 1.0.0, reshape 0.8.8, rlang 0.4.10, rstudioapi 0.13, rvest 0.3.6, splines 4.0.3, statmod 1.4.35, stringi 1.5.3, survival 3.2-7, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.6, withr 2.4.1, xfun 0.20, xml2 1.3.2