The "Happy Scientist" Workshop #1 An introduction to high-performance computing using R

George Vega Yon
vegayon@usc.edu

Garrett Weaver
gmweaver@usc.edu

USC Integrative Methods of Analysis for Genomic Epidemiology (IMAGE)
Department of Preventive Medicine
March 29, 2018

Agenda

1. High-Performance Computing: An overview
2. Parallel computing in R
3. Examples:
    1. *parallel*
    2. *iterators+foreach*
    3. *RcppArmadillo + OpenMP*
    4. *RcppArmadillo + OpenMP + Slurm*
4. Exercises

High-Performance Computing: An overview

Loosely, from R's perspective, we can think of HPC in terms of two, maybe three things:

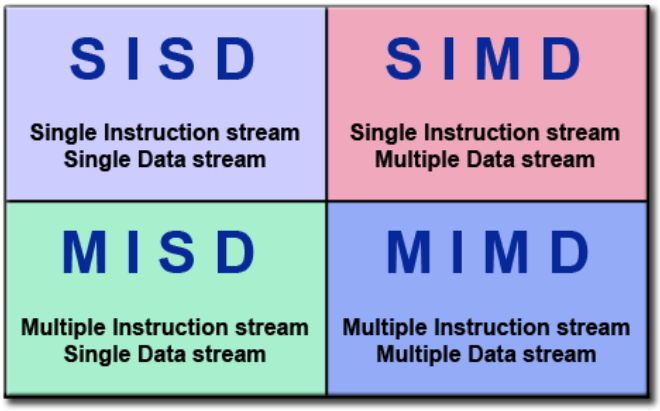
1. Big data: How to work with data that doesn't fit your computer
2. Parallel computing: How to take advantage of multiple core systems
3. Compiled code: Write your own low-level code (if R doesn't has it yet…)

(Checkout CRAN Task View on HPC)

Big Data

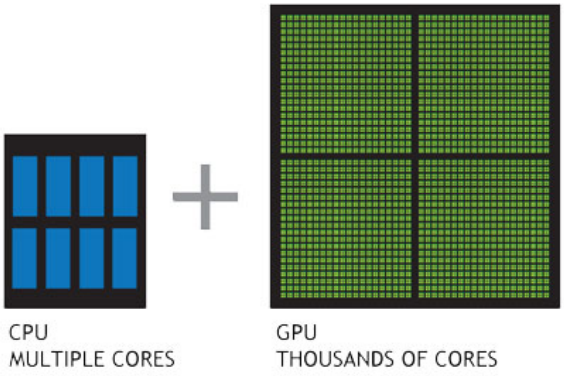
- Buy a bigger computer/RAM memory (not the best solution!)
- Use out-of-memory storage, i.e., don't load all your data in the RAM. e.g. The bigmemory, data.table, HadoopStreaming R packages
- Store it more efficiently, e.g.: Sparse Matrices (take a look at the `dgCMatrix` objects from the Matrix R package)

Parallel computing



Flynn's Classical Taxonomy (Introduction to Parallel Computing, Blaise Barney, Lawrence Livermore National Laboratory)

GPU vs CPU



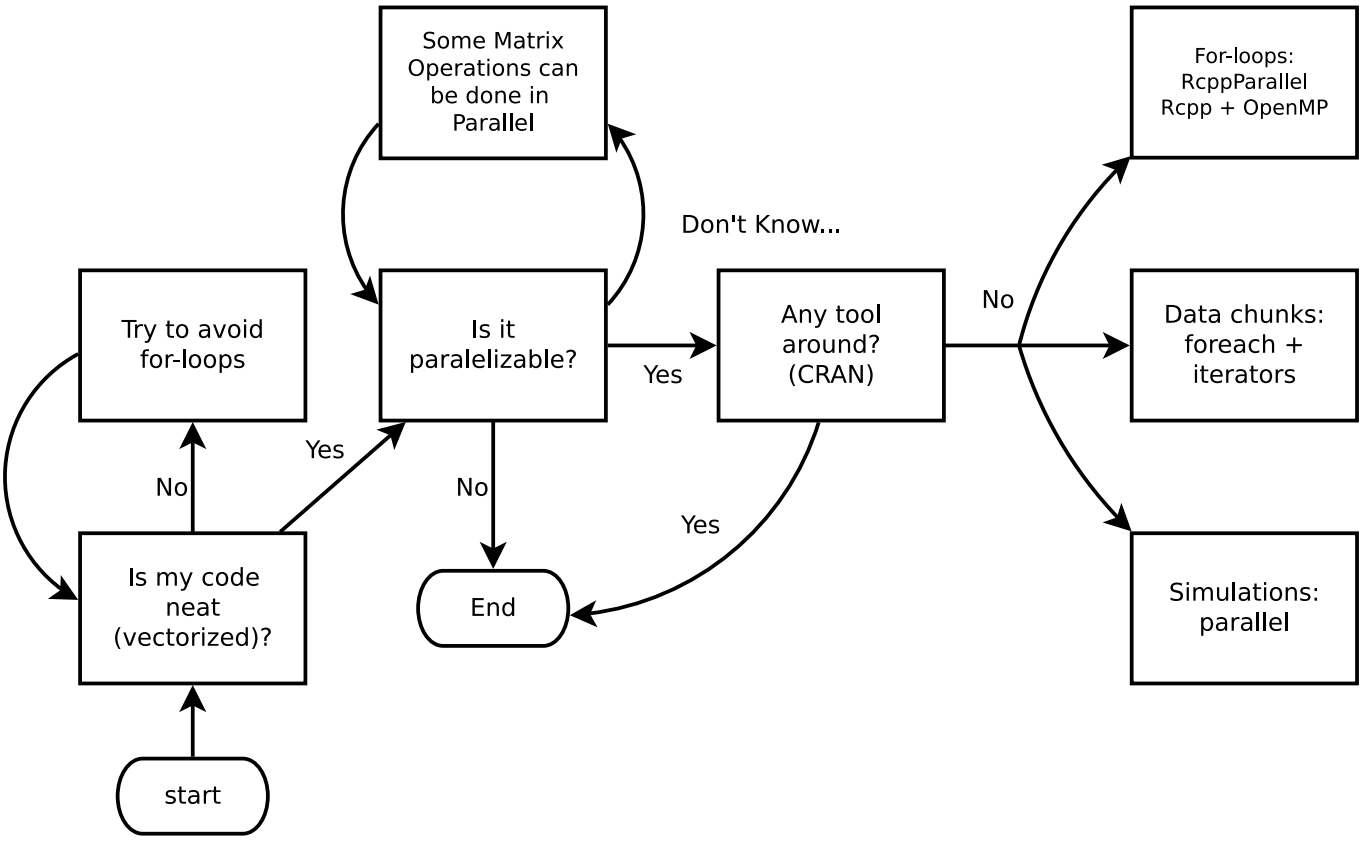
NVIDIA Blog

- **Why are we still using CPUs instead of GPUs?**

  GPUs have far more processor cores than CPUs, but because each GPU core runs significantly slower than a CPU core and do not have the features needed for modern operating systems, they are not appropriate for performing most of the processing in everyday computing. They are most suited to compute-intensive operations such as video processing and physics simulations. (bwDraco at superuser)

- Why use OpenMP if GPU is *suited to compute-intensive operations*? Well, mostly because OpenMP is **VERY** easy to implement (easier than CUDA, which is the easiest way to use GPU).

When is it a good idea?



Ask yourself these questions before jumping into HPC!

Parallel computing in R

While there are several alternatives (just take a look at the High-Performance Computing Task View), we'll focus on the following R-packages/tools for explicit parallelism:

1. R packages
    - ***parallel***: *R package that provides '[s]upport for parallel computation, including random-number generation'.*
    - ***foreach***: *'[A] new looping construct for executing R code repeatedly […] that supports parallel execution.'*
    - ***iterators***: *'tools for iterating over various R data structures'*
2. RcppArmadillo + OpenMP
    - ***RcppArmadillo***: *'Armadillo is a C++ linear algebra library, aiming towards a good balance between speed and ease of use.' '[RcppArmadillo] brings the power of Armadillo to R.'*
    - ***OpenMP***: *'Open Multi-Processing is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows.' (Wiki)*

Implicit parallelism, on the other hand, are out-of-the-box tools that allow the programmer not to worry about parallelization, e.g. such as **gpuR** for Matrix manipulation using GPU.

Parallel workflow

1. Create a cluster:
    1. *PSOCK Cluster: `makePSOCKCluster`: Creates brand new R Sessions (so nothing is inherited from the master), even in other computers!*
    2. *Fork Cluster: `makeForkCluster`: Using OS Forking, copies the current R session locally (so everything is inherited from the master up to that point). Not available on Windows.*
    3. *Other: `makeCluster` passed to **snow***
2. Copy/prepare each R session:
    1. *Copy objects with `clusterExport`*
    2. *Pass expressions with `clusterEvalQ`*
    3. *Set a seed*
3. Do your call:
    1. *`mclapply`, `mcmapply` if you are using **Fork***
    2. *`parApply`, `parLapply`, etc. if you are using **PSOCK***
4. Stop the cluster with `clusterStop`

parallel example 1: Parallel RNG

```
# 1. CREATING A CLUSTER
library(parallel)
cl <- makePSOCKcluster(2)

# 2. PREPARING THE CLUSTER
clusterSetRNGStream(cl, 123) # Equivalent to `set.seed(123)`

# 3. DO YOUR CALL
ans <- parSapply(cl, 1:2, function(x) runif(1e3))
(ans0 <- var(ans))
```

```
#               [,1]          [,2]
# [1,]  0.0861888293 -0.0001633431
# [2,] -0.0001633431  0.0853841838
```

```
# I want to get the same!
clusterSetRNGStream(cl, 123)
ans1 <- var(parSapply(cl, 1:2, function(x) runif(1e3)))

all.equal(ans0, ans1) # All equal!
```

```
# [1] TRUE
```

```
# 4. STOP THE CLUSTER
stopCluster(cl)
```

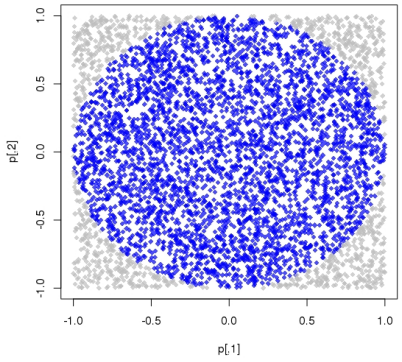
parallel example 1: Parallel RNG (cont.)

In the case of `makeForkCluster`

```
# 1. CREATING A CLUSTER
library(parallel)

# The fork cluster will copy the -nsims- object
nsims <- 1e3
cl    <- makeForkCluster(2)

# 2. PREPARING THE CLUSTER
RNGkind("L'Ecuyer-CMRG")
set.seed(123)

# 3. DO YOUR CALL
ans <- do.call(cbind, mclapply(1:2, function(x) {
  runif(nsims) # Look! we use the nsims object!
               # This would have fail in makePSOCKCluster
               # if we didn't copy -nsims- first.
  }))
(ans0 <- var(ans))

# Same sequence with same seed
set.seed(123)
ans1 <- var(do.call(cbind, mclapply(1:2, function(x) runif(nsims))))

ans0 - ans1 # A matrix of zeros

# 4. STOP THE CLUSTER
stopCluster(cl)
```

parallel example 2: Simulating π

- We know that $\pi = \frac{A}{r^2}$. We approximate it by randomly adding points x to a square of size 2 centered at the origin.
- So, we approximate π as $\Pr\{\|x\| \leq 1\} \times 2^2$



The R code to do this

```
pisim <- function(i, nsim) {  # Notice we don't use the -i-
  # Random points
  ans  <- matrix(runif(nsim*2), ncol=2)

  # Distance to the origin
  ans  <- sqrt(rowSums(ans^2))

  # Estimated pi
  (sum(ans <= 1)*4)/nsim
}
```

parallel example 2: Simulating π (cont.)

```
# Setup
cl <- makePSOCKcluster(10)
clusterSetRNGStream(cl, 123)

# Number of simulations we want each time to run
nsim <- 1e5

# We need to make -nsim- and -pisim- available to the
# cluster
clusterExport(cl, c("nsim", "pisim"))

# Benchmarking: parSapply and sapply will run this simulation
# a hundred times each, so at the end we have 1e5*100 points
# to approximate pi
rbenchmark::benchmark(
  parallel = parSapply(cl, 1:100, pisim, nsim=nsim),
  serial   = sapply(1:100, pisim, nsim=nsim), replications = 1
)[,1:4]
```

```
#       test replications elapsed relative
# 1 parallel            1   0.455    1.000
# 2   serial            1   1.842    4.048
```

```
ans_par <- parSapply(cl, 1:100, pisim, nsim=nsim)
ans_ser <- sapply(1:100, pisim, nsim=nsim)
stopCluster(cl)
```

```
#      par      ser        R
# 3.141561 3.141247 3.141593
```

The 'foreach' Package

- The 'foreach' package provides a looping construct to execute R code repeatedly in parallel
- The general syntax of `foreach` is similar to a standard for loop

```
# With parallelization --> %dopar%
output <- foreach(i = 'some object to iterate over', 'options') %dopar% {some r code}
# Without parallelization --> %do%
output <- foreach(i = 'some object to iterate over', 'options') %do% {some r code}
```

- As a first example, we use `foreach` without parallelization

```
result <- foreach(x = c(4, 9, 16, 25)) %do% sqrt(x)
result
```

```
# [[1]]
# [1] 2
#
# [[2]]
# [1] 3
#
# [[3]]
# [1] 4
#
# [[4]]
# [1] 5
```

- The default object returned is a list that contains the individual results compiled across all iterations

```
class(result)
```

```
# [1] "list"
```

Setting Up Parallel Execution With 'foreach'

- The steps to create the parallel backend are similar to the 'parallel' package

1. Create and register the cluster with the 'doParallel' package

    - *Method 1:* `makeCluster` + `registerDoParallel`

```
# Create cluster
myCluster <- makeCluster("# of cores", type = "PSOCK" or "FORK")
# Register cluster
registerDoParallel(myCluster)
```

    - *Method 2:* `registerDoParallel` + `cores` = *option*

```
# Create and register cluster
registerDoParallel(cores = "# of cores")
```

2. Use `foreach` with %dopar%

```
result <- foreach(x = c(4, 9, 16, 25)) %dopar% sqrt(x)
```

3. Stop the cluster (only required if you used Method 1 above)

```
stopCluster(myCluster)
```

- By default, Method 2 will create a SOCK cluster on Windows systems and a FORK cluster on Unix systems
- `getDoParWorkers()` can be used to verify the number of workers (cores)
- With some extra work and packages (Rmpi, doMPI), we can also use an MPI backend to enable parallelization on multiple nodes

The 'foreach' Package: Combining Results

- The `.combine` option is used to specify the function used to combine results

```
# Combine the results in a vector
foreach(x = c(4, 9, 16, 25), .combine = 'c') %dopar% sqrt(x)
```

```
# [1] 2 3 4 5
```

```
# Add the results together
foreach(x = c(4, 9, 16, 25), .combine = '+') %dopar% x
```

```
# [1] 54
```

- By default, `.combine` assumes that the function accepts two arguments (except for `c`, `cbind`, `rbind`)

```
# Custom combine that creates a running product
customCombine <- function(i, j) {
    cat(i, "\n")
    c(i, i[length(i)] * j)
}
foreach(x = c(2, 4, 6, 8, 10), .combine = customCombine) %dopar% x
```

```
# 2
# 2 8
# 2 8 48
# 2 8 48 384
```

```
# [1]    2    8   48  384 3840
```

- `.multicombine = TRUE` specifies that our combine function can accept more than two arguments
- `.maxcombine` sets the max number of arguments that can be passed to the combine function

foreach + iterators

- The 'iterators' package provides functions to generate: "A special type of object that supplies data on demand, one element at a time."
- `iter()` is a generic function to create iterators over common R objects (vector, list, data frame)

```
myIterator <- iter(object_to_iterate_over, by = "How to iterate over object", checkFunc = "optional functi
```

- Example: A simple vector iterator over odd integers

```
vector_iterator <- iter(c(5, 10, 15, 20), checkFunc = function(x) x %% 2 != 0)
```

```
str(vector_iterator)
```

```
# List of 4
#  $ state    :<environment: 0x3d2d748>
#  $ length   : int 4
#  $ checkFunc:function (x)
#   ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 1 55 1 77 55 77 1 1
#   .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x3dc5c58>
#  $ recycle  : logi FALSE
#  - attr(*, "class")= chr [1:2] "containeriter" "iter"
```

```
cat("obj:", vector_iterator$state$obj, " i:", vector_iterator$state$i, "\n")
```

```
# obj: 5 10 15 20  i: 0
```

```
cat("current element:", nextElem(vector_iterator), " i:", vector_iterator$state$i, "\n")
```

```
# current element: 5  i: 1
```

```
cat("current element:", nextElem(vector_iterator), " i:", vector_iterator$state$i, "\n")
```

```
# current element: 15  i: 3
```

foreach + iterators: Creating An Iterator

- An iterator that traverses over blocks of columns of a matrix

```
iblkcol <- function(mat, num_blocks) {
    # initialize variables to hold num of cols and current col index
    nc <- ncol(mat)
    i <- 1
    # define nextElem function
    next_element <- function() {
        if (num_blocks <= 0 || nc <= 0) stop('StopIteration') # stop when we have no more columns
        block_size <- ceiling(nc / num_blocks)                # determine block size
        col_idx <- seq(i, length = block_size)                # get indices
        i <<- i + block_size                                  # update current col index
        nc <<- nc - block_size                                # update num cols remaining
        num_blocks <<- num_blocks - 1                         # update num blocks remaining
        mat[, col_idx, drop = FALSE]                          # return next set of columns
    }
    # output list element with necessary class definitions
    itr <- list(nextElem = next_element)
    class(itr) <- c('iblkcol', 'abstractiter', 'iter')
    itr
}
myMatrix <- matrix(runif(200), nrow = 2, ncol = 100)
splitMatrix <- iblkcol(myMatrix, num_blocks = 25)
nextElem(splitMatrix)
```

```
#            [,1]      [,2]       [,3]      [,4]
# [1,] 0.1090886 0.9651406 0.01187498 0.8530483
# [2,] 0.8045434 0.7274180 0.08226454 0.5665807
```

- Note: The nextElem() function in splitMatrix is an example of a closure (encloses the environment of its parent function)

foreach Example: Bootstrapping

- Bootstrapping (Efron, 1979) uses the sample to learn about the sampling distribution of a statistic
  - *"The population is to the sample as the sample is to the bootstrap samples"*
- Nonparametric bootstrap
  - *To learn about characteristics of the sample, we sample (with replacement) from the original sample*
  - *Useful in scenarios where we do not want to assume a particular distribution for the statistic*
- Example pseudocode for a simple bootstrap to estimate the standard error for a sample statistic

```
# vector to hold estimates across bootstrap samples
est <- vector(mode = "numeric", length = Num_Bootstrap_Samples)

for (i in 1:Num_Bootstrap_Samples) {

    # Sample with replacement from original data (bootstrap sample size same as original data)
    boot_sample <- sample(1:n, n, replace = TRUE)

    # Compute statistic in bootstrap sample using some function g()
    est[i] <- g(dat[boot_sample])
}

# Compute standard error of estimate based on bootstrap samples
se_est <- sd(est)
```

foreach Example: Bootstrapping (Estimating a Median)

- Suppose we have the following sample of 100 observations for which we want to estimate the sample median

```
set.seed(123)
dat <- rnorm(n = 100, mean = 2.3, sd = 1.5)
dat[1:5]
```

```
# [1]  0.8471109  3.3591636  4.5335320 -0.4226388  2.7956144
```

- We can modify the previous code to run 10,000 bootstrap replicates and obtain an estimate of the standard error for the median

```
# vector to hold estimates across bootstrap samples
n <- length(dat)
num_reps <- 10000
median_est <- vector(mode = "numeric", length = num_reps)
for (i in 1:num_reps) {
    # Sample with replacement from original data (bootstrap sample size same as original data)
    boot_sample <- sample(1:n, n, replace = TRUE)
    # Compute statistic in bootstrap sample
    median_est[i] <- median(dat[boot_sample])
}
# Compute mean and standard error of median estimate based on bootstrap sample statistics
mean(median_est)
```

```
# [1] 1.820074
```

```
sd(median_est)
```

```
# [1] 0.7335949
```

foreach Example: Bootstrapping (Estimating a Median)

- The foreach loop is very similar

```
median_est_fe <- foreach(i = seq.int(num_reps), .combine = c) %dopar% {
    boot_sample <- sample(1:n, n, replace = TRUE)
    median(dat[boot_sample])
}
mean(median_est_fe)
```

```
# [1] 1.811854
```

```
sd(median_est_fe)
```

```
# [1] 0.7306325
```

- Timing: foreach is slower due to communication overheard and the low computational burden of the individual tasks

```
summary(microbenchmark::microbenchmark(
        for_loop = for (i in 1:num_reps) {
                            boot_sample <- sample(1:n, n, replace = TRUE)
                            median_est[i] <- median(dat[boot_sample])
                    },
        foreach = foreach(i = seq.int(num_reps), .combine = c) %dopar% {
                        boot_sample <- sample(1:n, n, replace = TRUE)
                        median(dat[boot_sample])
                  },
        times = 1)
    )[,c("expr","mean","median")]
```

```
#       expr      mean    median
# 1 for_loop  389.4132  389.4132
# 2  foreach 3288.3632 3288.3632
```

- To speed up, you may also consider "chunking" tasks to reduce communication overhead

foreach Example: Bootstrapping (Logistic Regression)

- As another example, we analyze the association between bacteria presence and treatment (drug vs. placebo) in the data set 'bacteria'
- To simplify the example, we sample both the outcome and predictors (i.e. the entire observation), this is known as case sampling

```
library(boot)
# get last observation point for each person (i.e. last time point tested for bacteria)
bact <- do.call(rbind, by(MASS::bacteria, MASS::bacteria$ID, function(x) x[which.max(x$week), ]))
# function to compute coefficents
boot_fun <- function(dat, idx, fmla) {
    coef(glm(fmla, data = dat[idx, ], family = binomial))
}
```

- Rather than write a for loop, the 'boot' package provides functions to easily apply our function across bootstrap replicates

```
set.seed(123)
boot(bact, boot_fun, R = 10000, fmla = as.formula(y ~ ap))
```

```
#
# ORDINARY NONPARAMETRIC BOOTSTRAP
#
#
# Call:
# boot(data = bact, statistic = boot_fun, R = 10000, fmla = as.formula(y ~
#     ap))
#
#
# Bootstrap Statistics :
#      original    bias    std. error
# t1* 0.7985077 0.0395186   0.4309512
# t2* 0.3646431 0.1520327   1.6066249
```

foreach Example: Bootstrapping (Logistic Regression)

- We can replicate the results from 'boot' with 'foreach'

```
set.seed(123)
n <- nrow(bact)
# generate bootstrap sample indices
indices <- sample.int(n, n * 10000, replace = TRUE)
dim(indices) <- c(10000, n)
# fit all bootstrap logistic regression models and extract coefficients into a matrix
results_foreach <- foreach(x = iter(indices, by = 'row'), .combine = cbind, .inorder = FALSE) %dopar% {
    coef(glm(y ~ ap, data = bact[x, ], family = binomial))
}
# standard error estimates
apply(results_foreach, 1, sd)
```

```
# (Intercept)         app
#   0.4309512   1.6066249
```

- The 'doRNG' package allows us to easily handle random number generation (note that the results will not match the previous)

```
library(doRNG)
results_foreach <- foreach(i = seq.int(10000), .combine = cbind, .inorder = FALSE) %dorng% {
    boot_samp <- sample.int(n, n, replace = TRUE)
    coef(glm(y ~ ap, data = bact[boot_samp, ], family = binomial))
}
apply(results_foreach, 1, sd)
```

```
# (Intercept)         app
#   0.4349333   1.6874428
```

foreach Example: Bootstrapping (Logistic Regression)

- Timing: With increased computational burden for each task, we see an improvement by using the parallel approach

```
summary(microbenchmark::microbenchmark(
        boot = boot::boot(bact, boot_fun, R = 10000, fmla = as.formula("y ~ ap")),
        foreach = foreach(i = seq.int(10000), .combine = cbind, .inorder = FALSE) %dopar% {
                          boot_samp <- sample.int(n, n, replace = TRUE)
                          coef(glm(y ~ ap, data = bact[boot_samp, ], family = binomial))
                  },
        times = 1, unit = 's')
    )[,c("expr","mean","median")]
```

```
#       expr      mean    median
# 1    boot 18.381147 18.381147
# 2 foreach  9.703243  9.703243
```

```
stopCluster(cl)
```

foreach example 2: Random Forests

- A number of statistical/learning methods involve computational steps that can be done in parallel
- Random forests, an ensemble method that involves generating a large number of decision trees, is one such example
  - *Each tree is generated based on a bootstrap sample from the original sample*
  - *Within each tree, each node/split of a decision tree is based on a random subset of variables (features)*
- Below we generate two classes and a set of potential predictors

```
# Random Forest Example

# Number of observations and predictor variables
n <- 1500
p <- 500

# Predictor data simulated as MVN(0,sigma) with AR(1) correlation
means_x <- rep(0, p)
var_x <- 1
rho <- 0.8
sigma_x <- matrix(NA, nrow = p, ncol = p)
for(i in 1:p){
    for(j in 1:p){
        sigma_x[i,j] <- var_x*rho^abs(i-j)
    }
}
X <- MASS::mvrnorm(n = n, mu = means_x, Sigma = sigma_x)

# Outcome is binary (two classes)
y <- gl(2, 750)
```

foreach example 2: Random Forests (cont.)

- Two changes in the call to `foreach`

```
# random forest sequentially
rf <- randomForest(X, y, ntree = 1000, nodesize = 3)

# random forest parallel (split up tree generation)
rf_par <- foreach(ntree = rep(250, 4),
                  .combine = combine,
                  .packages = "randomForest") %dopar% {
    randomForest(X, y, ntree = ntree, nodesize = 3)
}
```

  - *The 'randomForest' package already has a function to combine objects of class randomForest* `.combine = combine`
  - *The* `.packages` *option must be used to export the 'randomForest' package to all the cores*

- In the previous examples, we never explicitly export variables to the cores
  - *By default, (almost) all objects in the current scope that are refenced in* `foreach` *are exported*
  - `.export` *and* `.noexport` *can be used to control which objects are exported*

- Timing

```
#          expr     mean   median
# 1          rf 41.90121 41.90121
# 2 rf_parallel 11.57544 11.57544
```

RcppArmadillo and OpenMP

- Friendlier than **RcppParallel**… at least for 'I-use-Rcpp-but-don't-actually-know-much-about-C++' users (like myself!).
- Must run only 'Thread-safe' calls, so calling R within parallel blocks can cause problems (almost all the time).
- Use `arma` objects, e.g. `arma::mat`, `arma::vec`, etc. Or, if you are used to them `std::vector` objects as these are thread safe.
- Pseudo Random Number Generation is not very straight forward… But C++11 has a nice set of functions that can be used together with OpenMP
- Need to think about how processors work, cache memory, etc. Otherwise you could get into trouble… if your code is slower when run in parallel, then you probably are facing false sharing
- If R crashes… try running R with a debugger (see Section 4.3 in Writing R extensions):

```
~$ R --debugger=valgrind
```

RcppArmadillo and OpenMP workflow

1. Add the following to your C++ source code to use OpenMP, and tell Rcpp that you need to include that in the compiler:

```
#include <omp.h>
// [[Rcpp::plugins(openmp)]]
```

2. Tell the compiler that you'll be running a block in parallel with openmp

```
#pragma omp [directives] [options]
{
  ...your neat parallel code...
}
```

   You'll need to specify how OMP should handle the data:

   - `shared`: *Default, all threads access the same copy.*
   - `private`: *Each thread has its own copy (although not initialized).*
   - `firstprivate` *Each thread has its own copy initialized.*
   - `lastprivate` *Each thread has its own copy. The last value is the one stored in the main program.*

   Setting `default(none)` is a good practice.

3. Compile!

RcppArmadillo + OpenMP example 1: Distance matrix

```
#include <omp.h>
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(openmp)]]

using namespace Rcpp;

// [[Rcpp::export]]
arma::mat dist_par(arma::mat X, int cores = 1) {
  // Some constants
  int N = (int) X.n_rows;
  int K = (int) X.n_cols;

  // Output
  arma::mat D(N,N);
  D.zeros(); // Filling with zeros

  // Setting the cores
  omp_set_num_threads(cores);

#pragma omp parallel for shared(D, N, K, X) default(none)
  for (int i=0; i<N; i++)
    for (int j=0; j<i; j++) {
      for (int k=0; k<K; k++)
        D.at(i,j) += pow(X.at(i,k) - X.at(j,k), 2.0);

      // Computing square root
      D.at(i,j) = sqrt(D.at(i,j));
      D.at(j,i) = D.at(i,j);
    }


  // My nice distance matrix
  return D;
}
```

RcppArmadillo + OpenMP example 1: Distance matrix (cont.)

```
# Compiling the function
Rcpp::sourceCpp("dist.cpp")

# Simulating data
set.seed(1231)
K <- 5000
n <- 500
x <- matrix(rnorm(n*K), ncol=K)

# Are we getting the same?
table(as.matrix(dist(x)) - dist_par(x, 10)) # Only zeros
```

```
#
#       0
# 250000
```

```
# Benchmarking!
rbenchmark::benchmark(
  dist(x),                # stats::dist
  dist_par(x, cores = 1),  # 1 core
  dist_par(x, cores = 4),  # 4 cores
  dist_par(x, cores = 10), # 10 cores
  replications = 1, order="elapsed"
)[,1:4]
```

```
#                       test replications elapsed relative
# 4 dist_par(x, cores = 10)            1   0.512    1.000
# 3  dist_par(x, cores = 4)            1   1.180    2.305
# 2  dist_par(x, cores = 1)            1   2.358    4.605
# 1                 dist(x)            1   5.463   10.670
```

RcppArmadillo + OpenMP example 2: Simulating π

```
#include <omp.h>
#include <RcppArmadillo.h>
using namespace Rcpp;

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(openmp)]]

// C++11 provides several RNGs algorithms that can be set up to be thread safe.
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
double sim_pi(int m, int cores = 1, int seed = 100) {

  // Setting the cores
  omp_set_num_threads(cores);
  int n = m / cores;

  double ans = 0.0, d;
  double val = 4.0/m;
  double piest;
  int i;

#pragma omp parallel default(none) shared(ans, cores) \
  firstprivate(val, n, m, seed) \
  private(piest, i, d)
{

  // Which core are we
  int core_num = omp_get_thread_num();

  // Setting up the RNG
  // - The first line creates an engine that uses the 64-bit Mersenne Twister by
  //   Matsumoto and Nishimura, 2000. One seed per core.
  // - The second line creates a function based on the real uniform between -1
  //   and 1. This receives as argument the engine
  std::mt19937_64 engine((core_num + seed)*10);
  std::uniform_real_distribution<double> my_runif(-1.0, 1.0);

  double p0, p1;

  piest = 0.0;
  for (i = n*core_num; i < (n + n*core_num); i++) {

    // Random number generation (see how we pass the engine)
    p0 = my_runif(engine);
    p1 = my_runif(engine);

    d = sqrt(pow(p0, 2.0) + pow(p1, 2.0));

    if (d <= 1.0)
      piest += val;
  }

  // This bit of code is executed one thread at a time.
  // Instead of -atomic-, we could have use -critical-, but that has
  // a higher overhead.
```

```
  // a higher overhead.
#pragma omp atomic
  ans += piest;
}

  return ans;
}
```

RcppArmadillo + OpenMP example 2: Simulating π (cont.)

```
# Compiling c++
Rcpp::sourceCpp("simpi.cpp")

# Does the seed work?
sim_pi(1e5, cores = 4, seed = 50)
```

```
# [1] 3.1478
```

```
sim_pi(1e5, cores = 4, seed = 50)
```

```
# [1] 3.1478
```

```
# Benchmarking
nsim <- 1e8
rbenchmark::benchmark(
  pi01 = sim_pi(nsim, 1),
  pi04 = sim_pi(nsim, 4),
  pi10 = sim_pi(nsim, 10),
  replications = 1
)[,1:4]
```

```
#   test replications elapsed relative
# 1 pi01            1   2.578    6.095
# 2 pi04            1   0.915    2.163
# 3 pi10            1   0.423    1.000
```

~~No big speed gains… but at least you know how to use it now :)!~~ Nice speed gains!

RcppArmadillo + OpenMP + Slurm: Using the `rslurm` package

- The `rslurm` package (Marchand, 2017) provides a wrapper of Slurm in R.

- Without the need of knowing much about the syntax of `slurm`, this R package does the following:

  1. *Writes an R source file that sets up each node with your current config (packages, libpath, etc.). The outputs are stored in a known folder so these can be fetched out later.*

  2. *Writes a bash file with the call to `sbatch` (you can specify options).*

  3. *Executes the bash file and returns the jobid (you can query its status interatively).*

     Here is a simple example with our `sim_pi` function (so we are mixing OpenMP with Slurm!):

```
library(rslurm)

# How many nodes are we going to be using
nnodes <- 2L

# The slurm_apply function is what makes all the work
sjob <- slurm_apply(
  # We first define the job as a function
  f = function(n) {

    # Compiling Rcpp
    Rcpp::sourceCpp("~/simpi.cpp")

    # Returning pi
    sim_pi(1e9, cores = 8, seed = n*100)

  },
  # The parameters that `f` receives must be passed as a data.frame
  params        = data.frame(n = 1:nnodes), jobname = "sim-pi",

  # How many cpus we want to use (this when calling mcapply)
  cpus_per_node = 1,

  # Here we are asking for nodes with 8 CPUS
  slurm_options = list(`cpus-per-task` = 8),
  nodes         = nnodes,
  submit        = TRUE
)

# We save the image so that later we can use the `sjob` object to retrieve the
# results
save.image("~/sim-pi.rda")
```

Thanks!

```
# R version 3.4.3 (2017-11-30)
# Platform: x86_64-redhat-linux-gnu (64-bit)
# Running under: CentOS Linux 7 (Core)
#
# Matrix products: default
# BLAS/LAPACK: /usr/lib64/R/lib/libRblas.so
#
# locale:
#  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
#  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#  [9] LC_ADDRESS=C               LC_TELEPHONE=C
# [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#
# attached base packages:
# [1] parallel  stats     graphics  grDevices utils     datasets  methods
# [8] base
#
# other attached packages:
# [1] randomForest_4.6-14 doParallel_1.0.11   iterators_1.0.9
# [4] foreach_1.4.4
#
# loaded via a namespace (and not attached):
#  [1] Rcpp_0.12.16     codetools_0.2-15 digest_0.6.15    rprojroot_1.3-2
#  [5] backports_1.1.2  magrittr_1.5     evaluate_0.10.1  highr_0.6
#  [9] stringi_1.1.7    rmarkdown_1.9    tools_3.4.3      stringr_1.3.0
# [13] yaml_2.1.18      compiler_3.4.3   htmltools_0.3.6  knitr_1.20
```

Exercises

1. Generating multivariate normal random samples using parallel and foreach (`random-mvn.R` for pseudo-code, and `random-mvn-solution.R` for the implementation).

2. Fibonacci with Rcpp (`fib.R` for pseudo-code, and `fib-solution.R` for the implementation).

3. Rewriting the `scale` function using Rcpp and OpenMP (`scale.cpp` for pseudo-code, and `scale-solution.cpp` for the implementation).

Another example is provided in the file `wordcount.R` (you'll need the dataset `ulysses.txt`)

References

- Package parallel
- Using the iterators package
- Using the foreach package
- 32 OpenMP traps for C++ developers
- The OpenMP API specification for parallel programming
- 'openmp' tag in Rcpp gallery
- OpenMP tutorials and articles

For more, checkout the CRAN Task View on HPC