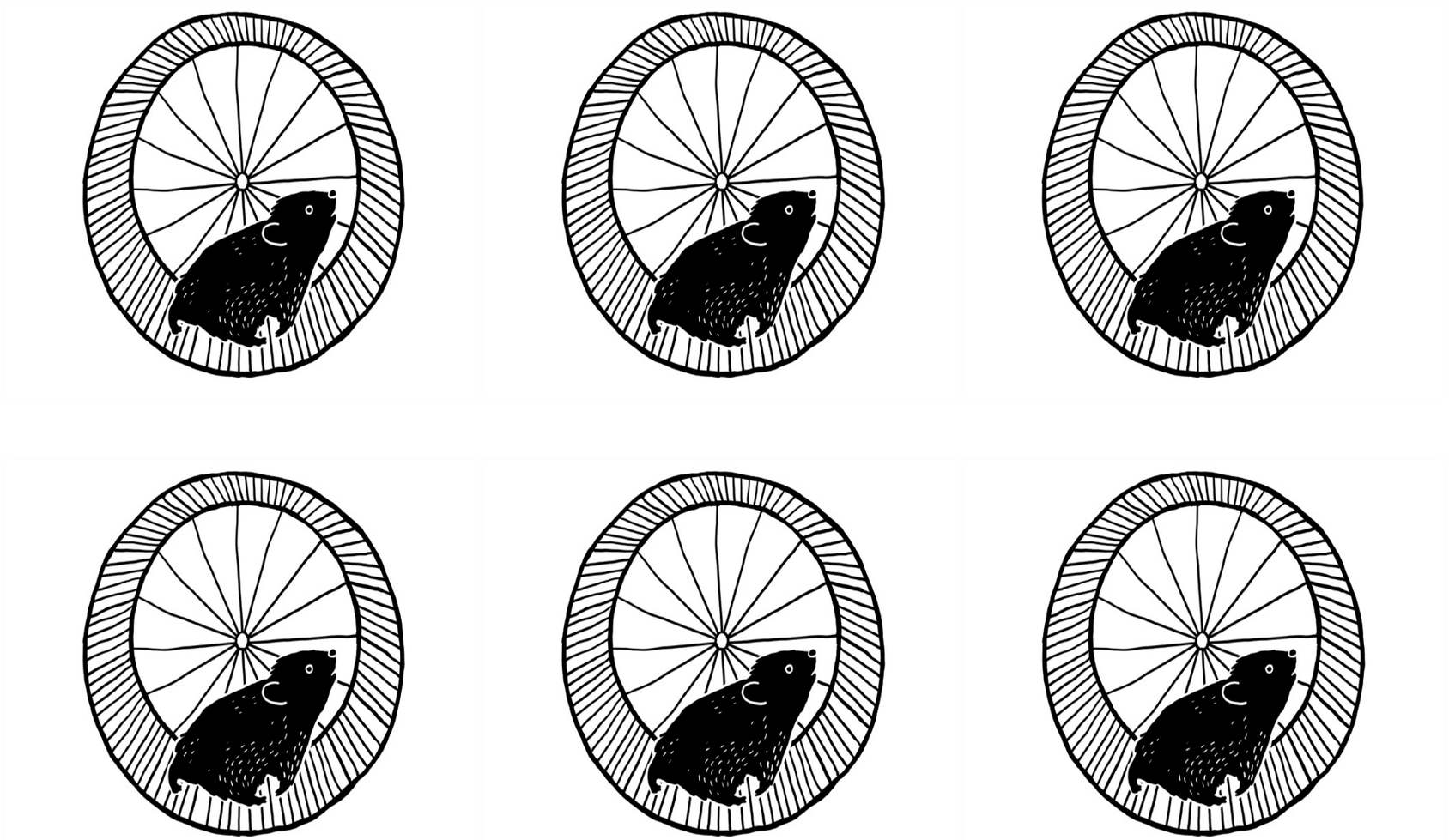
Parallel Computing in R

Christopher Grieves

What is Parallel Computing? (1)

- Splitting a problem into independent subproblems that can be solved simultaneously by taking advantage of multiple cores or CPU's.
- Ideally, having *p* processes means we can solve a problem *p* times faster.
- Above statement not necessarilly true due to overhead, varying job complexity, and hardware limitations.

What is Parallel Computing? (2)



”Embarrassingly Parallel”

- Term used to describe problem parallelized with little or no effort.
- Examples
  - Fitting models to subsets of data
  - MCMC Simulations
  - Bootstrap
  - K-Fold Cross-Validation

What do we use in R to do this? (1)

- We can use the **parallel** package included in R core to do this on a single machine with multiple processors/cores.
- We only consider parallel computing on a single machine with multiple cores.
- The package merges versions of two older packages together.
  - **SNOW** (Simple Network of Workstations)
  - **multicore**

What do we use in R to do this? (2)

- Other parallelization packages exist and even build on top of the **parallel** package. Examples:
  - **doParallel**/**foreach**
  - **snowfall**
- Some packages have a parallelization option built into it. Examples:
  - **boot**
  - **glmnet**

Terminology

- **Cluster** - a collection of processes launched by R (A.K.A. “workers”).
- **Master/Worker** - Master is the program that controls the cluster, Worker is the unit that responds to the Master process.
- **Core** - Hardware processing unit that receives instructions and does calculations.
  - A processor has mulitple cores which has multiple “hardware threads”. Each thread is a “logical” core in the eyes of R.

Two Types of Clusters (1)

Sockets

- Creates workers via **system("Rscript")**
- "Fresh" environment, variables need exporting and libraries reloaded.
- Must be used on windows, available on UNIX.

Two Types of Clusters (2)

Forking

- Makes an identical copy of the original process.
- Shares workspace, global options and loaded packages.
- No copy of memory used unless modified.
- Not Available on Windows.

Advisable to use this if you have access to Unix/Linux based machine.

Getting Started

First get number of cores on the machine.

```
library(parallel)
no_cores<-detectCores()
no_cores
```

```
[1] 4
```

Next we set up our “workers” by making our cluster. We use number of cores minus 1.

```
no_cores<-detectCores()-1

#make cluster uses SOCK by default but we
#explicitly pass it as an argument
cl<-makeCluster(no_cores, type="PSOCK")
summary(cl)
```

```
     Length Class    Mode
[1,] 3      SOCKnode list
[2,] 3      SOCKnode list
[3,] 3      SOCKnode list
```

Finally, Do Parallel Computing

```
library(parallel)
no_cores<-detectCores()-1

#make cluster uses SOCK by default but we
#explicitly pass it as an argument
cl<-makeCluster(no_cores, type="PSOCK")

#parallel version of lapply, note the
#extra argument of the cluster
results<-parLapply(cl,1:100,fun=sqrt)
class(results)
```

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

```
print(results[[3]])
```

```
[1] 1.732051
```

```
#finally clean up
stopCluster(cl)
```

!!We make sure to call **stopCluster(cl)** to stop worker processes and free memory!!

Parallel Analogs of R ”Apply” Functions

- Can be used as drop in replacements with very little effort.
- Analogs
  - lapply = parLapply
  - sapply = parSapply
  - apply = parApply

parLapply: Seasonal Package

```
library(parallel)
library(microbenchmark)
library(seasonal)

#Number of time series to simulate
N<-15

#Simulate Time Series
ts<-replicate(N,arima.sim(list(order = c(1,1,0),
ar = 0.7), n = 200))
ts.list<-lapply(split(ts,rep(1:ncol(ts), each =
nrow(ts))),
                function(x)
ts(x,frequency=12,start=c(2005,1)))

#Run in Sequence
microbenchmark(result.seq<-
lapply(ts.list,seas),times=1)
```

```
Unit: seconds
                                    expr
min       lq     mean   median
 result.seq <- lapply(ts.list, seas) 9.170651
9.170651 9.170651 9.170651
       uq      max neval
 9.170651 9.170651     1
```

```
#Run in Parallel
cl<-makeCluster(3)
microbenchmark(result.par<-
parLapply(cl,ts.list,seas),times=1)
```

```
Unit: seconds
                                          expr
min      lq    mean
 result.par <- parLapply(cl, ts.list, seas)
4.45783 4.45783 4.45783
  median      uq     max neval
```

- Seasonal Package was created to run Census' X-13 seasonal adjustment software and parse/visualize its results in R.
- We can use parLapply to parallelize the program and return the results in a list.
- microbenchmark package allows us to create timing statistics for R code over **n** runs.

For those folks not using Windows

- **mclapply** is easier version of parLapply on non-Windows Systems
- You supply number of cores, it runs everything for you in parallel.

```
library(parallel)
library(seasonal)

#Number of time series to simulate
N<-15

#Simulate Time Series
ts<-replicate(N,arima.sim(list(order = c(1,1,0),
ar = 0.7), n = 200))
ts.list<-lapply(split(ts,rep(1:ncol(ts), each =
nrow(ts))),
                        function(x)
ts(x,frequency=12,start=c(2005,1)))



#Run in Parallel
result<-mclapply(ts.list,seas,mc.cores = 15)
```

Let's pause for questions and to look at this overly happy Quokka.

When Is Parallel Computing Inefficient?

When overhead to create and communicate with parallel process takes longer than the computation it will be slower in than sequential.

- Methaphor
  - If I have workers and it takes longer for me to assign the task, complete it and return the results, then there is no gain in efficiency.

Inefficient Example

Calculating the square root of a vector of numbers.

```
#Create Vector of Numbers we want to apply function to
x<-1:100

#Get number of cores minus 1
no_cores<-detectCores()-1

#Set up your "Workers" to send tasks to
cl<-makeCluster(no_cores)
#cl<-makeCluster(2)

#We run the process in parallel and in sequence to compare the results
microbenchmark(sqrt.par<-parLapply(cl,x,sqrt),times=100)
```

```
Unit: microseconds
                                  expr    min       lq     mean   median
 sqrt.par <- parLapply(cl, x, sqrt) 628.45 680.9515 762.7053 718.2565
       uq      max neval
 811.2205 2428.923   100
```

```
stopCluster(cl)

#Run sequentially
microbenchmark(sqrt.seq<-lapply(x,sqrt),times=100)
```

```
Unit: microseconds
                        expr    min    lq     mean median    uq    max
 sqrt.seq <- lapply(x, sqrt) 39.476 40.66 45.73301  41.45 49.74 87.241
 neval
   100
```

Cluster Level Functions

Sometimes we wish to manually call functions on clusters. We turn to some of the following functions below to lend us a hand:

clusterEvalQ clusterSetRNGStream

clusterExport

- Challenges these functions help solve
  - Exporting objects from master to workers
  - Loading required libraries on workers.
  - Ensuring unique random seeds.

clusterEvalQ(cl,expression)

- Evaluates **expression** on every worker in the cluster.
- Can be used for example to load necessary libraries on each worker.
- Returns a list of the results, but can be discarded in this case.

```
cl<-makeCluster(3)
clusterEvalQ(cl,library(MASS))
```

```
[[1]]
[1] "MASS"      "methods"   "stats"
"graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[2]]
[1] "MASS"      "methods"   "stats"
"graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[3]]
[1] "MASS"      "methods"   "stats"
"graphics"  "grDevices" "utils"
[7] "datasets"  "base"
```

```
stopCluster(cl)
```

clusterExport(cl, varlist)

- Exports variables (objects) in varlist from master to workers in cluster **cl**
- When using socket cluster, variables must be exported to child processes.

```
cl<-makeCluster(3)
x<-1:10
clusterExport(cl,c("x"))
clusterEvalQ(cl,sum(x))
```

```
[[1]]
[1] 55

[[2]]
[1] 55

[[3]]
[1] 55
```

```
stopCluster(cl)
```

clusterSetRNGStream(cl, iseed)

- Function sets random seeds on each worker in a reproducible fashion.
- With forking, every process gets the same seed so we have to do this.

```
library(parallel)

cl<-makeCluster(3,type="FORK")

#Generate 1 normal random number on each cluster
with same seed
clusterEvalQ(cl,rnorm(1))
```

```
[[1]]
[1] 1.027703

[[2]]
[1] 1.027703

[[3]]
[1] 1.027703
```

```
#set random seed on each worker
clusterSetRNGStream(cl,iseed = 1000)

#Generate 1 normal random number on each cluster
with unique
clusterEvalQ(cl,rnorm(1))
```

```
[[1]]
[1] -0.4634739

[[2]]
[1] 0.5713205

[[3]]
[1] 0.4065615
```

```
#stop cluster
stopCluster(cl)
```

Interactive Examples

- K-Fold Cross Validation
- Bootstrap
- MCMC simulations

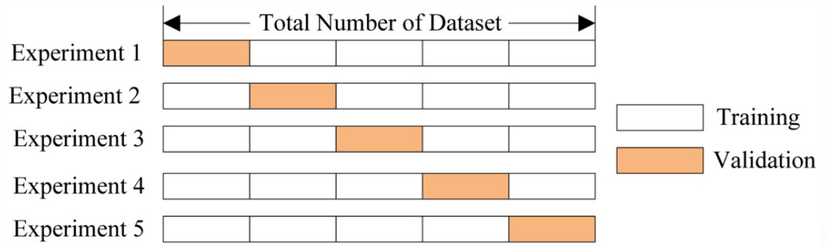
K-Fold Cross Validation

Model validation technique

This is usually used to evaluate how well a model performs and to choose tuning parameters.

This can be easily parallelized because we can give each of the K folds to a separate worker to evaluate.

1. Divide dataset up into K equal portions.
2. Remove 1-Kth of the dataset.
3. Train model on remaining dataset.
4. Evaluate trained model on removed portion.
5. Repeat steps 2-4 for all K portions of the data.



Non-Parametric Bootstrap

Suppose we wish to perform inference on a parameter $\hat{\theta} = s(X)$ from sample:

$$X = (X_1, X_2, \ldots, X_n)$$

We will draw B (usually a pretty large number) independent "bootstrap" samples by sampling from the original dataset with replacement.

$$X^{(1)}, \ldots, X^{(B)}$$

We then create estimates of theta and perform inference on those.

$$\theta^{(b)} = s(X^{(b)})\ b = 1, \ldots, B$$

We parallelize this by performing the sampling and creating the bootstrap samples and estimates in parallel.

MCMC Simulations

- For complicated distributions with no analytical solution, we turn to sampling methods to generate a sample.
- Gibbs Sampler, Metropolis Hastings Algorithm. Uses a Markov Chain to sample from a distribution.
- We can run these chains in parallel to generate a large population
- I use JAGS (Just Another Gibbs Sampler)
- RStan is another sampler, it has parallel built into it.

Closing Remarks

- **parLapply** and **mclapply** are R functions that can make parallelization pretty simple.
- Parallelization is not a silver bullet for faster computation, some care is needed to get better results.
- Be kind if you are using a server… other people also use it.
- For the more complicated aspects, I have created "Appendix" slides that cover advanced topics (i.e. pedantic and boring). These can be very helpful though.

Thank You Everyone

Chris Grieves

SMS-CES

grieves.christopher@bls.gov

Appendix Slides

- Load Balancing
- Copy-on-Modify with Forking
- Comparing Socket Vs Fork Cluster Performance

Load Balancing

- parallel package preassigns tasks to workers.
- “Load Balancing” allows for dynamically assigning tasks at the expense of more overhead.
- Example shows difference when jobs have varying amounts of time to complete. Load Balancing improves our time if some tasks take much longer than others.

```
  #Lets create a contrived example where Load
Balancing matters. We simply make the system sleep
for a fixed amount of time.In our example we use
3 workers and assign them wait times
  #of c(1,..,1,10,1,...,1)
  wait.times<-rep(1,10*3)
  wait.times[21]<-10
  print(wait.times)
```

```
 [1]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1 10  1  1
[24]  1  1  1  1  1  1  1
```

```
  cl<-makeCluster(3,type="FORK")

  #In this case since jobs are preassigned, one
cluster will take 19 seconds to finish (10+9*1).
No other cores help after they finish theres which
take 10 seconds
  system.time(results.par<-
clusterApply(cl,wait.times,Sys.sleep))
```

```
   user  system elapsed
  0.006   0.002  19.027
```

```
  #In this case, we dynamically assign the wait
times thus once the workers with an easy
assignment are finished, they help to finish the
remaining jobs.
  system.time(results.par<-
clusterApplyLB(cl,wait.times,Sys.sleep))
```

```
   user  system elapsed
  0.006   0.002  16.020
```

Copy-on-Modify with Forking

- With forking children are a copy of master process.
- Care must be taken when working with large datasets.
- Modifying a variable from the master causes the memory to be copied.
- To the right, you can see that all variables hold the same memory address until modified, hence copied.

```
library(parallel)
library(pryr)
x<-1
#original address of variable x
address(x)
```

```
[1] "0x199af98"
```

```
cl<-makeCluster(2,type="FORK")
#address of x on cluster processes
clusterEvalQ(cl,c(as.character(x),address(x)))
```

```
[[1]]
[1] "1"         "0x199af98"

[[2]]
[1] "1"         "0x199af98"
```

```
#modify x and check addresses
clusterApply(cl,2:3,function(y)
{c(as.character(x<<-y),address(x))})
```

```
[[1]]
[1] "2"         "0x154d618"

[[2]]
[1] "3"         "0x18838f8"
```

Comparing Socket Vs Fork Cluster Performance

Example to illustrate FORK vs PSOCK performance. We create a random linear model with 70 variables and 8000 observations below and Normal random noise with mean 0 and sigma=50.

```
#packages
library(microbenchmark)
library(parallel)

#data matrix parameters
#we will create data by simulation
num.vars<-70
num.rows<-8000

#Linear Model Noise (Normal noise with mean 0)
sigma<-50

#Create Variable Names
vars<-paste0("X",(1:num.vars))

#create coeficients randomly
coef<-sample(10:500,num.vars,replace=FALSE)

#create data matrix
DF <-as.data.frame(matrix(runif(num.rows*num.vars,-10,10),
        nrow=num.rows,
        ncol = num.vars,
        dimnames=list(NULL,vars)))

#create response variable and add noise
DF$y <- (as.matrix(DF) %*% coef) + rnorm(n = num.rows,0,sigma)
```

Functions and parameters for cross val

```
# Global Variables
#######################################################################

#create formula for model creation
form<-as.formula(paste0("y~", paste0(vars,collapse ="+" ),sep=""))

#Cross Validation Parameter for number of folds
K<-10

#####################################################################
# cross.val.one(i)
#
# Evaluates the mean square error of prediction in the i-th of K
# cross validation folds and returns the mean square error
# for that fold.
#####################################################################
cross.val.one<-function(i) {
  #logical vector with data to leave out
  leave.out<- as.logical(1:nrow(DF) %% K == (i-1))

  y<-DF[leave.out,]$y  #True values left out set

  mdl<-lm( formula = form, data=DF[!leave.out,])

  y.hat<-predict(mdl,DF[leave.out,]) #Predict on left out set

  return(mean( (y.hat-y)^2))
}

microbenchmark(results.seq<-sapply(1:K,cross.val.one),times=5,unit="s")
```

Done Sequentially (without parallelization)

```
Unit: seconds
                                        expr      min       lq     mean
 results.seq <- sapply(1:K, cross.val.one) 1.087932 1.142265 1.218671
   median      uq      max neval
 1.225899 1.23938 1.397877     5
```

Done in parallel with socket cluster:

```
go_sock<-function() {
  #Start Cluster
  cl<-makeCluster(10,type="PSOCK")

  #Send data, number of folds, and the formula
  clusterExport(cl,c("DF","K","form"))

  #Perform Cross Validation on each fold
  results.par.sock<-
parSapply(cl,1:K,cross.val.one)

  #Stop cluster
  stopCluster(cl)
  return(results.par.sock)
}
#Benchmark results
microbenchmark(go_sock(),times=5,unit="s")
```

```
Unit: seconds
       expr      min       lq     mean   median
uq      max neval
 go_sock() 2.308489 2.335357 2.337819 2.339422
2.349276 2.356552     5
```

Done in parallel with fork cluster:

```
go_fork<-function() {
  #Start Cluster
  cl<-makeCluster(10,type="FORK")

  #Not needed for Forking
  #clusterExport(cl,c("DF","K","form"))

  #Perform Cross Validation on each fold
  results.par.fork<-
parSapply(cl,1:K,cross.val.one)

  #Stop cluster
  stopCluster(cl)
  return(results.par.fork)
}
#Benchmark results
microbenchmark(go_fork(),times=5,unit="s")
```

```
Unit: seconds
       expr       min        lq      mean
median        uq      max
 go_fork() 0.2830106 0.2833765 0.2939398 0.2969468
0.3002981 0.306067
 neval
     5
```

The End