

Parallel and high performance processing with R

An introduction to the high performance computing environment at SHARCNET

Ge Baolai

SHARCNET

Western University

- *Running R on SHARCNET*
- *Running R many simulations at once*
- *Parallel processing with R*
- *Other aspects of HPC with R*

General Interest Seminar Series
Teaching the lab skills for
SCIENTIFIC COMPUTING

THE ART OF R PROGRAMMING

A TOUR OF STATISTICAL SOFTWARE DESIGN

NORMAN MATLOFF

Use R!

Christian P. Robert
George Casella

**Introducing Monte
Carlo Methods with R**

 Springer

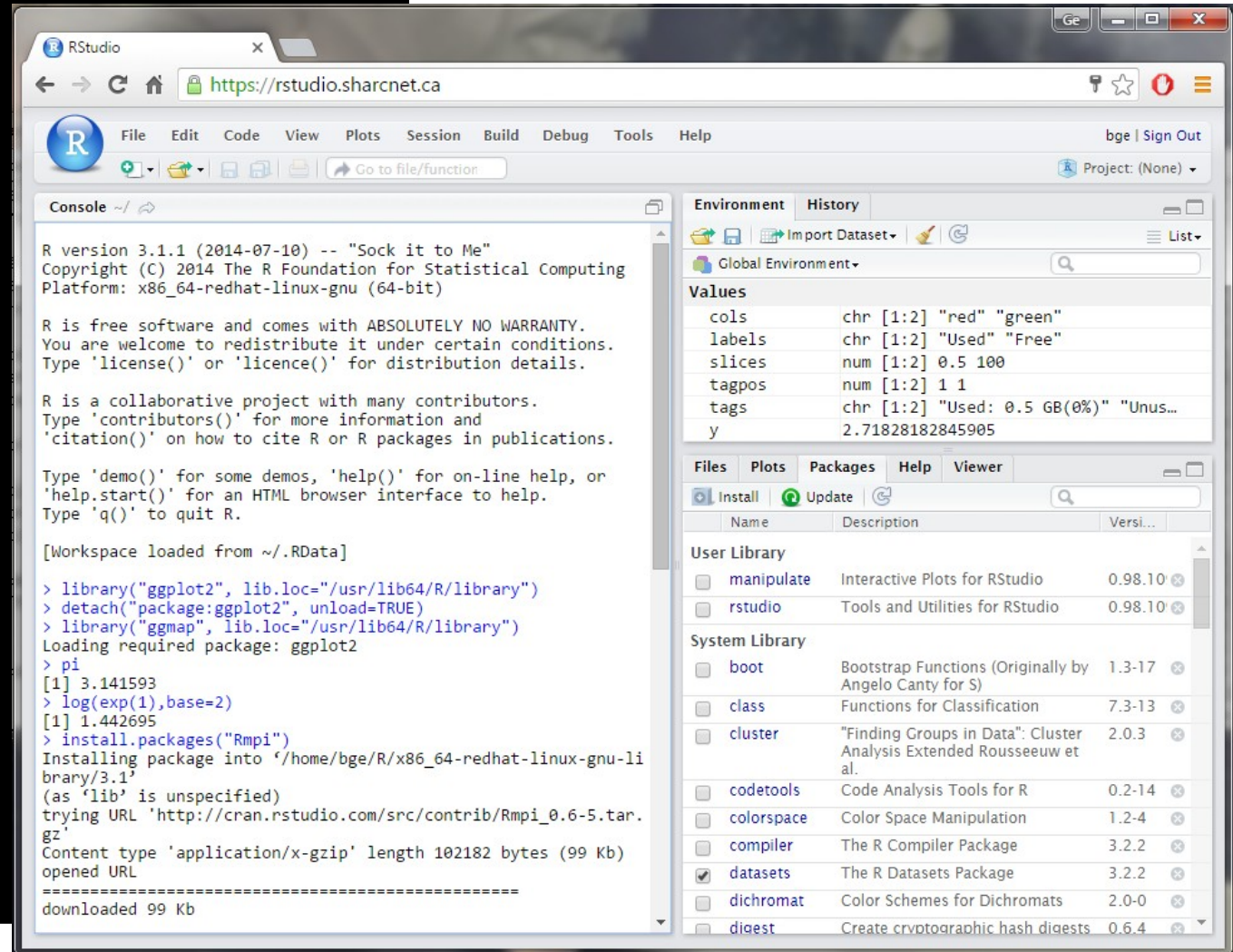
 no starch
press

On clusters

Online

<http://rstudio.sharcnet.ca/>

```
$ module unload intel
$ module load r
$ R
```



Example: I am to run 10 simulations, each can go independently. I'd like to run them on SHARCNET systems as 10 independent jobs, by typing the command time times:

```
$ sqsub -r 3d -o sim1.log R CMD BATCH --no-save --args param1.csv sim.R  
$ sqsub -r 3d -o sim2.log R CMD BATCH --no-save --args param1.csv sim.R  
$ sqsub -r 3d -o sim3.log R CMD BATCH --no-save --args param3.csv sim.R  
$ sqsub -r 3d -o sim4.log R CMD BATCH --no-save --args param4.csv sim.R  
$ sqsub -r 3d -o sim5.log R CMD BATCH --no-save --args param5.csv sim.R  
$ sqsub -r 3d -o sim6.log R CMD BATCH --no-save --args param6.csv sim.R  
$ sqsub -r 3d -o sim7.log R CMD BATCH --no-save --args param7.csv sim.R  
$ sqsub -r 3d -o sim8.log R CMD BATCH --no-save --args param8.csv sim.R  
$ sqsub -r 3d -o sim9.log R CMD BATCH --no-save --args param9.csv sim.R  
$ sqsub -r 3d -o sim10.log R CMD BATCH --no-save --args param10.csv sim.R  
$ sqjobs
```

Powered by Linux

But if I need to run 300 simulations, then typing commands 300 times becomes impractical. Instead I'd write a BASH script, say, “run_sims.sh” to automate that:

```
#!/bin/bash
num_sims=300
for ((i=1;i<$num_sims;i++)); do
    sqsub -r 3d -o sim$i.log R CMD BATCH --no-save --args param$i.csv sim.R
done
```

Then I run the script

```
$ ./run_sims
$ sqjobs
```

The 300 jobs are now in the queue. The scheduler will find free cores and place the jobs on them at a later time.

Example (cont'd): Suppose I have 80 simulations, each uses an input file with irregular name, e.g. patient name, SmithKW.csv, JohnFK.csv, WarrenB.csv, how do I automate the submissions?

```
#!/bin/bash
for f in *.csv; do
    sqsub -r 3d -o $f.log R CMD BATCH --no-save --args $f sim.R
done
```

Running R on multicores

```
sqsub -q threaded -n 8 -mpp=4g -o myprog.log R CMD BATCH --no-save myprog.R
```

Running R across nodes (via MPI)

```
sqsub -q mpi -n 32 -mpp=4g -o myprog.log R CMD BATCH --no-save myprog.R
```

We won't talk much about R+MPI (Rmpi) here. Bottom line: tell the scheduler how many MPI process you want to run, and never spawn dynamic MPI processes from within your code without telling the scheduler at the time of submission.

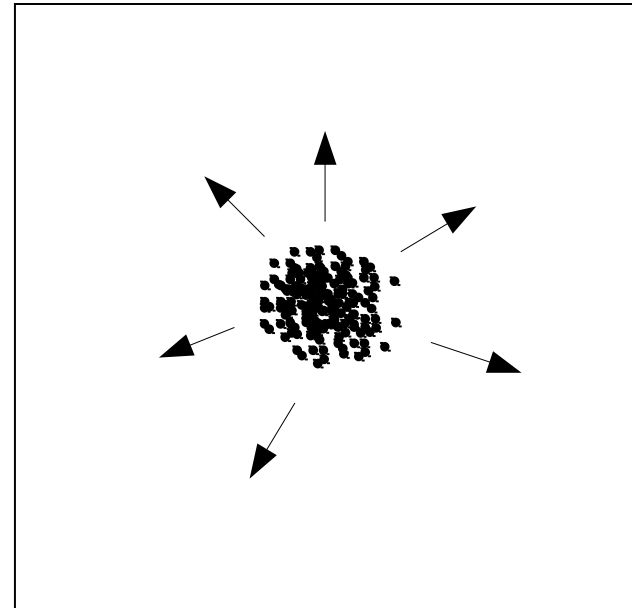
Simulation of diffusion process

- Substance of particles at the centre at the beginning.
- To simulate the distribution of the particles over time.

Assumptions:

- Each particle – the *walker* – walks randomly independent of other.
- *Each one walks a small distance over a small, unit time step.*
- *At each point, the probability of a walker arriving at this location depends only on the equal probability of it having reached the neighboring points.*

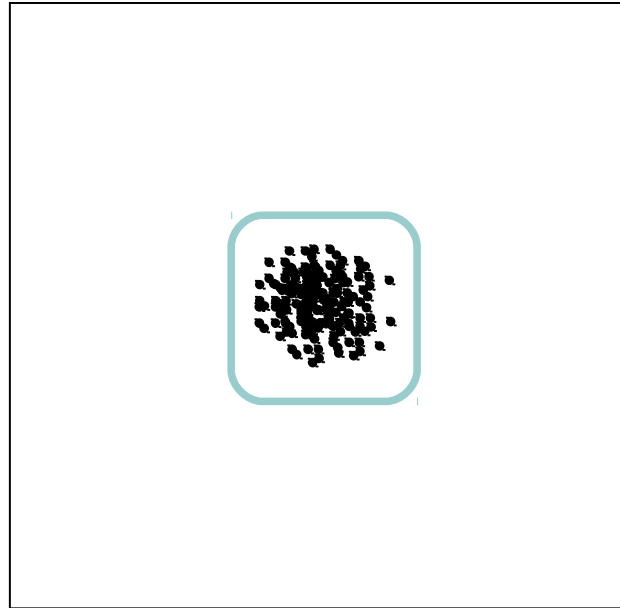
2D



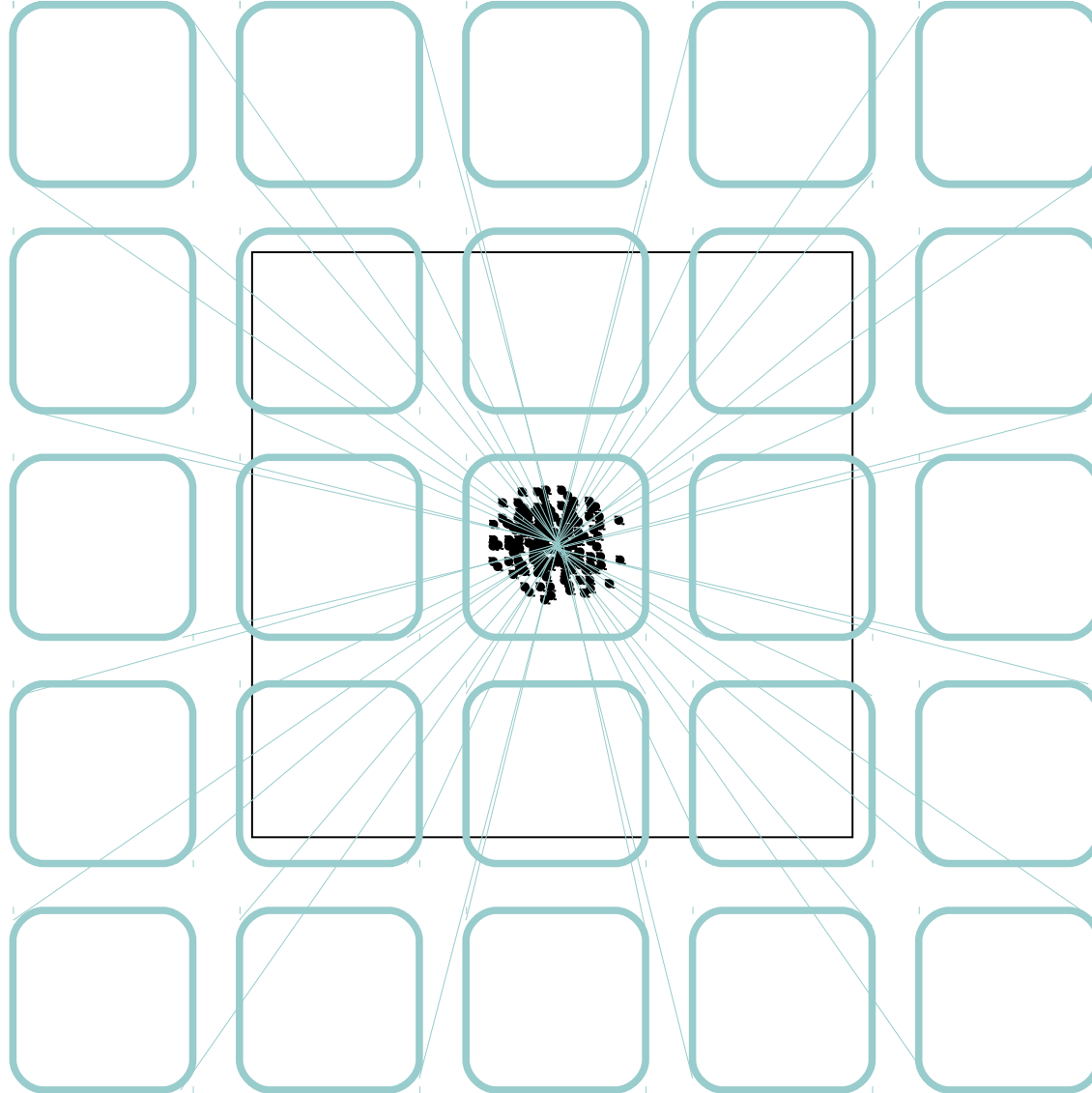
1D



- On a single core – Use **for** loop to iterate through walkers.



- On multicores – Use **foreach**, each core follows a subset of walkers.



Parallel packages

There are many parallel packages:- (that enable one to perform parallel processing from least to advanced levels, including, e.g.

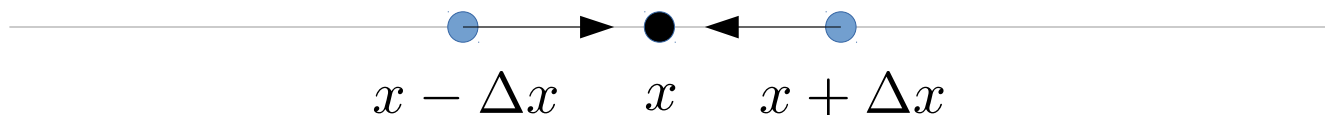
- **multicore** – enables the use of all cores on a single computer. It uses `fork()`, a Unix mechanism, to spawn multiple instances, not for Windows.
- **snow** – Simple Network Of Workstations, can run on a single computer and a cluster of computers (nodes), works for both Windows and Linux.
- **parallel** – built on top of **multicore** and **snow**, now part of R base package.
- **foreach** – a package that enables one to perform parallel for loops.
- Rmpi, Rdsn, pbdR, etc.

Exercise: Simulation of 1D diffusion process

Assumptions:

- All particles start at the origin.
- Each particle – the *walker* – walks randomly, either leftward or rightward, with equal probability, independent of other.
- Each one walks a distance Δx over a small, unit time step Δt .
- At each point, the probability of a walker arriving at this location depends only on the equal probability of it having reached the neighboring points, that is

$$p(x, t + \Delta t) = \frac{1}{2}p(x - \Delta x, t) + \frac{1}{2}p(x + \Delta x, t).$$



Implementation 1 (inefficient, never do this)

```
num_walkers = 100000
```

```
num_paths = 200
```

```
x = matrix(0,num_paths,num_walkers)
```

```
x2 = rep(0,num_paths)
```

Launch random walkers, all starting from x = 0

```
set.seed(47)
```

```
ts <- proc.time()
```

```
for (i in 1:num_walkers)
```

```
{
```

A walker completes its walk

```
for (k in 2:num_paths)
```

```
{
```

```
  #x[k,i] = x[k-1,i] + sample(c(-1,1))[1]
```

```
  x[k,i] = x[k-1,i] + rnorm(1,0,1)
```

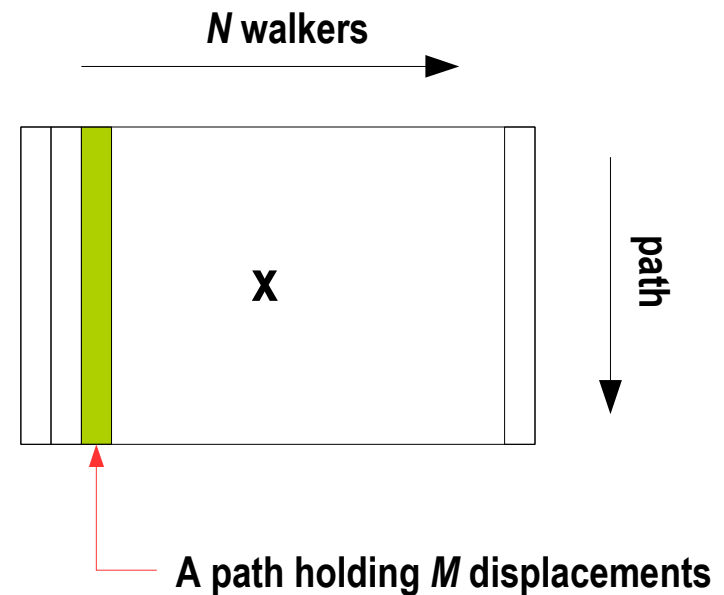
```
}
```

```
}
```

```
proc.time() - ts
```

Use two arrays:

- x – paths, in column.
- x2 – variance of displacements.
- Note, R stores arrays column major. So data access should be by column too.



```
# Implementation 1 (inefficient, never do this)
```

```
num_walkers = 100000
```

```
num_paths = 200
```

```
x = matrix(0,num_paths,num_walkers)
```

```
x2 = rep(0,num_paths)
```

```
# Launch random walkers, all starting from x = 0
```

```
set.seed(47)
```

```
ts <- proc.time()
```

```
for (i in 1:num_walkers)
```

```
{
```

```
  # A walker completes its walk
```

```
  for (k in 2:num_paths)
```

```
  {
```

```
    #x[k,i] = x[k-1,i] + sample(c(-1,1))[1]
```

```
    x[k,i] = x[k-1,i] + rnorm(1,0,1)
```

```
  }
```

```
}
```

```
proc.time() - ts
```

```
# Compute the variance
```

```
for (k in 1:num_paths)
```

```
{
```

```
  x2[k] = sum(x[k,]*x[k,])/num_walkers
```

```
}
```

```
# Plot a path
```

```
plot(x[,1],type='l',xlab='Steps',ylab='Displacement')
```

```
# Plot the variance
```

```
plot(1:num_paths,x2,xlab='Displacement',ylab='Variance');
```

```
# Plot the distribution of displacements at last step
```

```
hist(x[num_paths,],freq=TRUE)
```

```
save(x,x2,file="vars.RData")
```

Implementation 2 – using foreach + parallel packages

library(foreach)

library(doParallel) # parallel and iterator loaded implicitly

num_walkers = 100000

num_paths = 200

x2 = rep(0,num_paths)

Launch random walkers, all starting from x = 0

set.seed(47) **See L'Ecuyer generator (1999)**

ts <- proc.time()

registerDoParallel(4)

result <- foreach (i=1:num_walkers) %dopar%

{

 for (k in 2:num_paths) # A walker completes its walk

 {

 x2[k] = x2[k-1] + **rnorm**(1,0,1)

 }

 return(x2)

}

stopImplicitCluster()

proc.time() - ts

Assemble the result to path matrix

x <- matrix(unlist(result),num_paths,num_walkers)

Compute the variance

for (k in 1:num_paths)

{

 x2[k] = sum(x[k,]*x[k,])/num_walkers

}

Plot a path

plot(x[,1],type='l',xlab='Steps',ylab='Displacement')

Plot the variance

plot(1:num_paths,x2,xlab='Displacement',ylab='Variance');

Plot the distribution of displacements at last step

hist(x[num_paths,],freq=TRUE)

save(x,x2,file="vars.RData")

Implementation 3 – no parallelism, just using R functions

```
num_walkers = 100000
```

```
num_paths = 200
```

```
x = matrix(0,num_paths,num_walkers)
```

```
x2 = rep(0,num_paths)
```

```
# Launch random walkers, all starting from x = 0
```

```
set.seed(47)
```

```
ts <- proc.time()
```

```
for (i in 1:num_walkers)
```

```
{
```

```
  disp = rnorm(num_paths,mean=0,1)
```

```
  x[,i] = cumsum(disp)
```

```
}
```

```
proc.time() - ts
```

Compute the variance and generate plots.

Vectorization

Notice for each walker, the displacements from the origin are

$$x_{i+1} = x_i + \Delta x_i.$$

This cumulative sum can be completed efficiently by one shot using R's function `cumsum()`. Compare with

```
for (i in 1:num_walkers)
```

```
{
```

```
  for (k in 2:num_paths)
```

```
  {
```

```
    x[k,i] = x[k-1,i] + rnorm(1,0,1)
```

```
  }
```

```
}
```

Exercise (cont'd): Performance comparison

num_walkers	for loop (sec)	foreach (sec) on 4 cores	for loop + cumsum() (sec)
1,000	2.011	0.630	0.068
10,000	19.327	4.567	0.722
100,000	195.000	50.895	6.740

Using Rmpi – *Explicit parallel programming with MPI*

- Developed by Prof. Yu Hao from Western University.
- To gain the fine grained control, use direct message passing send/receive calls featured by the message passing interface MPI.
- Offers greater flexibility for implementing complex algorithms, than many other parallel packages.
- There is a learning curve, if not already knowing MPI.
- Requires system installation of MPI.
- Not so straightforward to setup compared to other packages.

Using Rmpi on SHARCNET

- Load **gcc** and **gcc compiled OpenMPI** module
- Load R module
- Set environment **R_LIBS**, e.g. to \$HOME/lib/R
- Install Rmpi from within R
- Copy R to \$HOME/bin/R, add the following lines (**red**) at line 4

```
#!/bin/sh  
# Shell wrapper for R executable.
```

```
PATH=$MPI_ROOT/bin:$PATH; export PATH  
LD_LIBRARY_PATH=$MPI_ROOT/lib:$LD_LIBRARY_PATH  
export R_PROFILE=$R_LIBS/Rmpi/Rprofile
```

```
... ..
```

Example

```
module unload intel openmpi  
module load r  
module load gcc/5.1.0 openmpi/gcc-5.1.0/std/1.8.7  
sqsub -q mpi -n 8 -r 10m -o yu.log \  
$HOME/bin/R CMD BATCH --no-save yu.R
```

```
library(Rmpi)
```

```
#setup parallel random number generator
```

```
mpi.setup.rngstream()
```

```
#create your own function(s)
```

```
myfun=function(n) mean(rnorm(n))
```

```
#transfer your function(s) to all slaves
```

```
mpi.bcast.Robj2slave(myfun)
```

```
#run the parallel job
```

```
output <- mpi.parReplicate(1000,myfun(1000000))
```

```
output[1:10]#can save output to a file
```

```
#must close all slaves
```

```
mpi.close.Rslaves()
```

```
mpi.quit()
```

Vectorization

Using R functions **xapply()** to perform operations on a list of things at once can make computations really fast.

- **lapply(x, fun, ...)** – apply a function to each element of a list/vector
- **sapply(x, fun, ...)** – apply a function to each element of a list/vector and simplify to return a vector or array.
- **vapply(x, fun, fun_value, ...)** – Tips: same as sapply, but returns a vector of type matching fun_value (safe); if the length of fun_value==1, then it returns a vector of the same length of **x**. This will be faster (don't know exactly why).
- **tapply** – apply a function to a slice of list, vector, easier for data frames.
- **mapply** – a multivariate version of apply().
- **apply(x, margin, fun, ...)** – apply a function to a row, column or elements of an array, with margin==1 being rows and 2 being columns.

lapply/sapply(x, fun, ...) – *passing one argument*

```
# Pass ONE argument to the function
```

```
> n <- c(2,3,5)
```

```
> x <- lapply(n,rnorm)
```

```
> x
```

```
[[1]]
```

```
[1] 0.6766938 -1.3893758
```

```
[[2]]
```

```
[1] -1.7145366 -2.4362372 0.2003453
```

```
[[3]]
```

```
[1] -1.7807025 -0.1330609 -0.2210980 -0.1071721 -0.2836180
```

```
> y <- sapply(x,mean)
```

```
> y
```

```
[1] -0.3563410 -1.3168095 -0.5051303
```

lapply/sapply(x, fun, ...) cont'd – *passing multiple arguments*

Pass ONE argument to the function

```
ns = c(2,3,5)
x = lapply(ns,rnorm)
y = sapply(x, mean)
```

Pass TWO or more arguments to the function?

This doesn't work

```
path <- function(n, x0=0, dev=1) { ds = rnorm(n,mean=x0,sd=dev); return(cumsum(c(x0,ds[1:n-1]))); }
y = sapply(1:5, path(n=5,x0=1,dev=1))
```

This works, but not so obvious

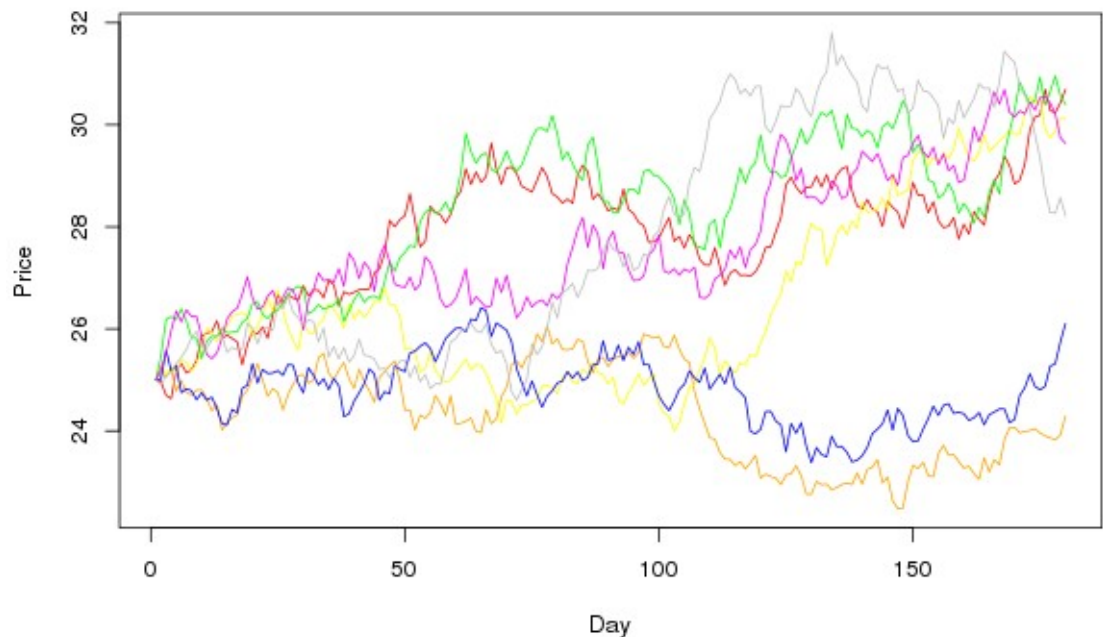
```
path <- function(i,n, x0=0, dev=1) { ds = rnorm(n,mean=x0,sd=dev); return(cumsum(c(x0,ds[1:n-1]))); }
y = sapply(1:5, path, n=5,x0=1,dev=1)
```

This works too, at least consistent to the function definition

```
path <- function(n, x0=0, dev=1) { ds = rnorm(n,mean=x0,sd=dev); return(cumsum(c(x0,ds[1:n-1]))); }
y = sapply(1:5, function(n,x0,dev) path(n=5,x0=1,dev=1))
```

Exercise: Simulating stock prices

- To simulate the closing price at the end of **180 days**.
- Assume the stock price follows the normal distribution (??) on a daily basis.
- Assume an average of 0.1% of gain of its opening price (e.g. **\$25**), and a volatility of 0.001.
- To generate **100,000** scenarios (paths) of movements and examine the results at the end of **180 days**.



```
# Stock price simulation - serial version
stock_prices <- function(price,ndays,gain=0,sigma=0)
{
  ds = 1+rnorm(ndays-1,mean=gain,sd=sigma)
  return(cumprod(c(price,ds)))
}

set.seed(47)
system.time(prices <- replicate(100000,
  stock_prices(price=25,
    ndays=180,
    gain=0.001,
    sigma=0.01)))

paths=matrix(unlist(prices),nrow=180,ncol=100000)
ps = sample.int(num_paths,min(num_paths,7))
pmin =min(paths[,ps])
pmax = max(paths[,ps])
plot(paths[,ps[1]],type='l',col='red',xlab='Day',ylab='Price',ylim
=c(pmin,pmax))
```

Vectorization

Assume the stock price follows a normal distribution (well, not really).

Let q_t be the change rate in stock price, the new price is given by

$$S_{t+1} = q_t S_t, \quad t = 1, \dots, N$$

We use R function **rnorm()** to generate a vector of change rates and **cumprod()** to generate a vector of prices over time in one shot.

Then we use function **replicate()** to repeat the process 100,000 times to generate 100,000 paths.

Vectorization is fast!

Stock price simulation - parallel version

library(parallel)

stock_prices <- function(price,ndays,gain=0,sigma=0)

{

 s <- .Random.seed

 nextRNGStream(s)

 set.seed(s)

 ds = **rnorm**(ndays-1,mean=1+gain,sd=sigma)

 return(**cumprod**(c(price,ds)))

}

See L'Ecuyer generator (1999)

RNGkind("L'Ecuyer-CMRG")

set.seed(47)

system.time(prices <- **mclapply**(1:100000,

function(price,ndays,gain,sigma)

 stock_prices(price=25,

 ndays=180,

 gain=0.001,

 sigma=0.01),mc.cores=4))

paths=matrix(unlist(prices),nrow=180,ncol=100000)

Stock price simulation - serial version

stock_prices <- function(price,ndays,gain=0,sigma=0)

{

 ds = **rnorm**(ndays-1,mean=1+gain,sd=sigma)

 return(**cumprod**(c(price,ds)))

}

set.seed(47)

system.time(prices <- **replicate**(100000,

 stock_prices(price=25,

 ndays=180,

 gain=0.001,

 sigma=0.01)))

paths=matrix(unlist(prices),nrow=180,ncol=100000)

Implicit parallelization

Featured by the underlying libraries, no work needed, free.

```
# In "mm.R"  
n = 4*1024  
n2 = n*n  
A = matrix(rnorm(n2),nrow=n,ncol=n)  
B = matrix(rnorm(n2),nrow=n,ncol=n)  
system.time(C <- A %*% B)
```

Set the environment variable **OMP_NUM_THREADS** to different values and run the script, see the execution time difference.

```
$ export OMP_NUM_THREADS=1  
$ R --no-save < mm.R  
  
$ export OMP_NUM_THREADS=2  
$ R --no-save < mm.R  
  
$ export OMP_NUM_THREADS=4  
$ R --no-save < mm.R
```

Powered by Linux

When things can bite...

Unexpected behavior may occur when using parallel packages

Perfectly correct, but troublesome R code!!!

```
> library(parallel)
> set.seed(1000)
> test <- lapply(1:10,function(x) rnorm(100000))
> system.time(x <- mclapply(test,function(x) loess.smooth(x,x), mc.cores=1))
  user system elapsed 
2.968 0.026 2.991 
> system.time(x <- mclapply(test,function(x) loess.smooth(x,x), mc.cores=2))
```



This code is correct, but troublesome. It may suffer from that

- The code does not scale at all
- The code hangs

Why?

Large datasets and linear models

Look for alternatives

```
# To fit data with linear model y = b0 + b1*x + b2*x^2
```

```
n = 5000000
```

```
p = 2
```

```
x = sort(runif(n,-1,1))
```

```
y = sin(x)*exp(x) + rnorm(n,sd=0.25)
```

```
# Create a linear model (quadratic polynomial). This may fail!
```

```
system.time(m <- lm(y ~ x + I(x^2)))
```

```
m
```

```
# Try this one if the above fails – equivalent
```

```
t = proc.time()
```

```
A = outer(x,0:p,'^')
```

```
coef <- qr.solve(A,y)
```

```
proc.time() - t
```

```
coef
```

```
# And even try this one via the solution of normal equations - not recommended
```

```
t = proc.time()
```

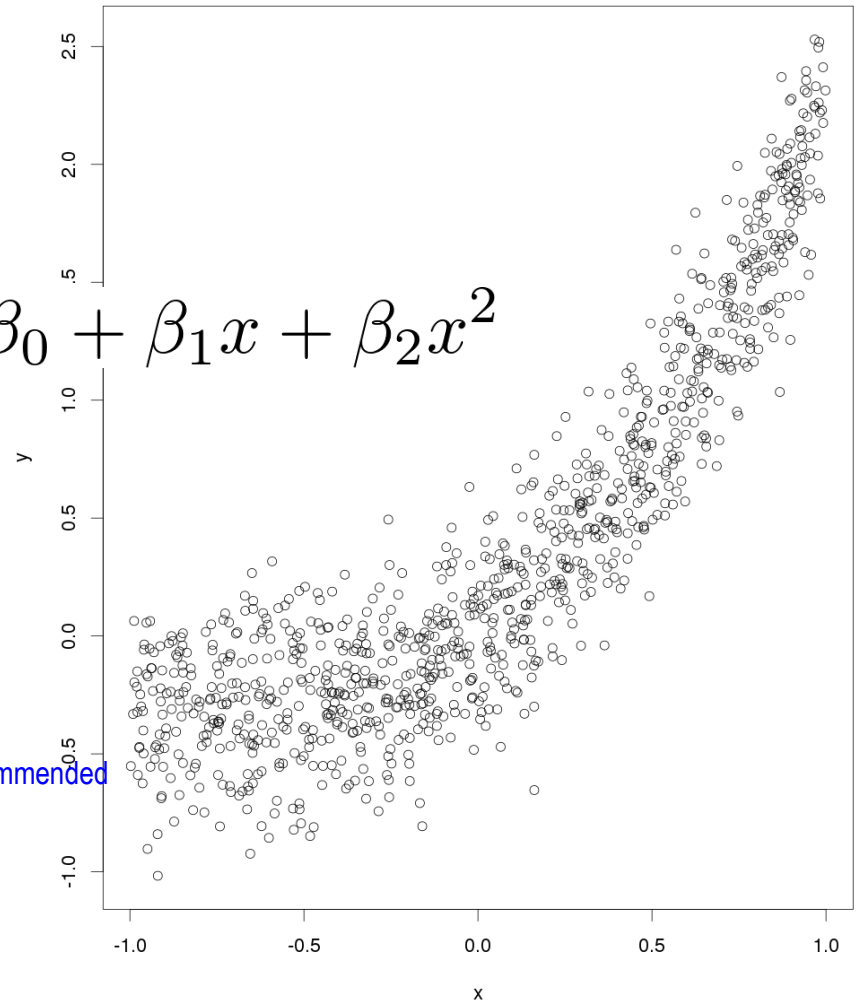
```
B = t(A) %*% A
```

```
coef2 <- solve(B, t(A) %*% y)
```

```
proc.time() - t
```

```
coef2
```

$$y = \beta_0 + \beta_1 x + \beta_2 x^2$$



Loading large CSV files

I have a large CSV file containing data extracted from a database, 350MB of size, 1.7 millions of records, 28 columns each. The job data file contains the following

- Number of cores used
- Arrival, start time and end time, etc.

```
> system.time(j <- read.csv("jobs_orca.csv",header=T,sep=', '))
```

```
   user  system elapsed  
318.522   1.778  325.615
```

>5 min

```
> j[sample.int(120000,6),c("ncpus","t_in","t_start")]
```

	ncpus	t_in	t_start
90163	1	2014-12-12 05:25:35-05	2014-12-12 05:27:36-05
94375	1	2014-12-12 12:38:51-05	2014-12-12 17:13:46-05
13681	16	2014-11-20 20:07:26-05	2014-11-20 20:11:28-05
37321	1	2014-11-27 01:02:35-05	
89417	1	2014-12-12 02:52:07-05	2014-12-12 02:53:58-05
48207	1	2014-11-28 16:27:02-05	2014-11-28 16:27:59-05

← Missing data

Using R function **read.csv()** takes nearly 6 minutes to load the data on my laptop running Windows 7. So, how to improve this?

WELCOME!

[Follow @rbloggers](#) 25.1K

Here you will find daily news and tutorials about R, contributed by over 573 bloggers.

There are many ways to follow us -
By e-mail:

Your e-mail here

Subscribe

 21571 readers
BY FEEDBURNER

On Facebook:


 R blogg...
28k likes

[Like Page](#)

Be the first of your friends to like this



If you are an R blogger yourself you are invited to add your own R content feed to this site (Non-English R bloggers should add themselves- here)

[JOBS FOR R-USERS](#)

 SHRM Temporary -
Certification @

Efficiency of Importing Large CSV Files in R

February 10, 2014

By statcompute

[Like](#)
[Share](#)

7

[Tweet](#)
[Share](#)

2

(This article was first published on [Yet Another Blog in Statistical Computing](#) » S+/R, and kindly contributed to R-bloggers)

```

1  ### size of csv file: 689.4MB (7,009,728 rows * 29 columns)
2
3  system.time(read.csv('../data/2008.csv', header = T))
4  #   user  system elapsed
5  # 88.301   2.416   90.716
6
7  library(data.table)
8  system.time(fread('../data/2008.csv', header = T, sep = ";"))
9  #   user  system elapsed
10 #  4.740    0.048    4.785
11
12 library(bigmemory)
13 system.time(read.big.matrix('../data/2008.csv', header = T, sep = ";"))
14 #   user  system elapsed
15 # 59.544    0.764   60.308
16
17 library(ff)
18 system.time(read.csv.ffdf(file = '../data/2008.csv', header = T, sep = ";"))
19 #   user  system elapsed
20 # 60.028    1.280   61.335
21
22 library(sqldf)
23 system.time(read.csv.sql('../data/2008.csv'))
24 #   user  system elapsed
25 # 87.461    3.880   91.447

```

Add a Comment

[Like](#)
[Share](#)

7

[Tweet](#)
[Share](#)

2

TOP 3 POSTS FROM THE PAST 2 DAYS

Cleaning and visualizing genomic data: a case study in tidy analysis
Installing R packages
In-depth introduction to machine learning in 15 hours of expert videos

TOP 9 ARTICLES OF THE WEEK

1. Installing R packages
2. James Bond movies
3. In-depth introduction to machine learning in 15 hours of expert videos
4. Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance
5. Using apply, sapply, lapply in R
6. Correlation and Linear Regression
7. Online R courses at Udemy - for (only) \$11
8. Deep Learning with MXNetR
9. How to Make a Histogram with Basic R

SPONSORS

||||| |||||

MANGOSOLUTIONS



R Consulting, Training, Support and Application Development

Loading large CSV files (cont'd)

I use package `data.table`, it loads data much faster!

```
> library(data.table)
> system.time(d <- fread("jobs_orca.csv"))
Read 1635034 rows and 28 (of 28) columns from 0.323 GB file in 00:00:22
  user  system elapsed
14.24   0.34   21.65
```

Next, how should I do to get the following?

- Sorted by number of cores used
- The min, max, mean and median wait time, etc grouped by number of cores.

People used to ***procedural programming languages*** may get lost. R is better at this sort of things.

Using aggregate functions

I use package data.table, it loads data much faster!

```
> library(data.table)
> system.time(d <- fread("jobs_orca.csv"))
Read 1635034 rows and 28 (of 28) columns from 0.323 GB file in 00:00:22
  user  system elapsed
 14.24   0.34   21.65
> names(d)
[1] "jobid"      "host"      "state"      "job_type"
[5] "t_in"       "t_start"   "t_end"      "utime"
[9] "stime"      "atime"     "ncpus"      "nnodes"
[13] "exitstatus" "memory"    "pfaults"    "flags"
[17] "nodes"      "institution" "user"       "est_runtime"
[21] "pi_user"    "exit_info"  "queue_type_id" "pvmem_req"
[25] "vmem"       "vmem_req"   "gpus"       "backfilled"
```

Next, how should I do to get the following?

- Sorted by number of cores used
- The min, max, mean and median wait time, etc grouped by number of cores.

People used to ***procedural programming languages*** may get lost. R is better at this sort of things.

Using aggregate functions (cont'd)

```
> library(data.table)
> system.time(d <- fread("jobs_orca.csv"))
Read 1635034 rows and 28 (of 28) columns from 0.323 GB file in 00:00:22
   user  system elapsed
 14.24    0.34   21.65
> ds <- subset(d,select=c(as.numeric(ncpus),t_in,t_start,t_end))
> d_cpus <- aggregate(ds$ncpus,by=list(ds$ncpus),FUN =length)
> names(d_cpus) <- c("ncpus","jobs")
> d_cpus[order(as.numeric(d_cpus$ncpus)),]
   ncpus    jobs
1      1 1351160
31     2   45262
55     4   52676
80     8   88775
22    16   41459
39    24    2990
51    32   20180
54    36    1361
63    48    1171
71    64   17101
13   128     603
...
45   256     189
```


Large data and out of core operations

I have a 1.45GB job data (a subset of 5 year's data), 34 million records, that can't *easily* fit in the memory.

```
> library(data.table)
> > system.time(d <- fread('jobs.csv'))
Read 31629152 rows and 6 (of 6) columns from 1.350 GB file in 00:00:42
  user  system elapsed
 21.45    2.48   42.07
> gc()
      used (Mb) gc trigger (Mb) max used (Mb)
Ncells  552184 29.5   940480  50.3   940480  50.3
Vcells 95811153 731.0 115596418 882.0 95815259 731.1
> names(d)
[1] "jobid"  "sysid"  "ncpus"  "t_in"   "t_start" "t_end"
> system.time(wt <- d$t_start-d$t_in)
  user  system elapsed
 0.20    0.06    0.28
> system.time(quantile(wt, probs=0.75))
  user  system elapsed
 0.56    0.03    0.64
```

← After

Large data and out of core operations (cont'd)

I have a 1.45GB job data (a subset of 5 year's data), 34 million records, that can't *easily* fit in the memory. I use ***bigmemory*** package.

```
> library(bigmemory)
> gc(reset=TRUE)
```

	used (Mb)	gc trigger (Mb)	max used (Mb)
Ncells	527579 28.2	940480 50.3	527579 28.2
Vcells	886272 6.8	1650153 12.6	886272 6.8

← Before

```
>
> system.time(j <-
read.big.matrix('jobs.csv',header=T,backingfile='jobs.bin',descriptorfile='jobs.desc'))
  user  system elapsed
239.68   40.54  291.08
```

Warning message:

```
In read.big.matrix("jobs.csv", header = T, backingfile = "jobs.bin", :
  Because type was not specified, we chose integer based on the first line of data.
```

```
> gc()
```

	used (Mb)	gc trigger (Mb)	max used (Mb)
Ncells	534454 28.6	940480 50.3	557062 29.8
Vcells	898302 6.9	1650153 12.6	965413 7.4

← After

Large data and out of core operations (cont'd)

Creating file-backed big matrix off disk is slow, but saves a lot memory. Operations on the data are pretty fast.

```
> library(bigmemory)
> system.time(d <-
read.big.matrix('ts.csv',header=T,backingfile='ts.bin',descriptorfile='ts.desc'))
  user  system elapsed
239.68   40.54   291.08
> dd <- dget('ts.desc')
> d <- attach.big.matrix(dd)
> system.time(wt <- d[, "t_start"] - d[, "t_in"])
  user  system elapsed
 0.31    0.08    0.39
> system.time(quantile(wt, probs=0.75))
  user  system elapsed
 0.53    0.04    0.56
> system.time(wt_min <- min(wt))
  user  system elapsed
 0.06    0.00    0.05
```

More workshops

- Introduction to Unix shell (software carpentry)
- Revision control with Git (software carpentry)
- Programming with Python (software carpentry)
- Introduction to R (software carpentry)
- Parallel programming with R (software carpentry)
- Introduction of SQL database (software carpentry)
- Introduction to parallel computing with MATLAB
- Introduction to parallel computing with modern Fortran
- Bi-weekly online seminars: <https://www.sharcnet.ca/my/news/calendar>
- Summer school on high performance and scientific computing

Slides

<http://www.sharcnet.ca/~bge/seminars/parallel-R/parallel-hpc-R.pdf>

Acknowledgment

Some materials in this talk were taken from the course “Introduction to R” at 2015 Ontario summer school on HPC at University of Toronto, July 13-17, given by Erik Spence of SciNet, University of Toronto.

Find where we are

Western Science Centre, Room 127

Shared Hierarchical Academic Computing Network (SHARCNET)

Western University

Web: <http://www.sharcnet.ca/>

E-mail: help@sharcnet.ca