



Parallel Computing with R using snow and snowfall

Vinh Q. Nguyen

Department of Statistics
University of California, Irvine

November 18, 2009



Outline

- 1 Introduction
Motivation
- 2 snow and snowfall
Setup
Examples



Outline

- 1 Introduction
Motivation
- 2 snow and snowfall
Setup
Examples



R

- S is a great language for
 - data exploration
 - data analysis
 - model fitting
 - graphics
- R is relatively easy to learn
- Very extensible
 - Convenient for research
 - Many statistical packages are available
- R is notoriously slow for computationally intensive tasks
 - well, for *new* tasks, and
 - loops



Ways to speed up R

- Vectorize your code
 - think in terms of vectors
 - not always possible for complicated code
- Interface to compiled code (C/C++/Fortran)
 - learning curve is very high (on top of knowing R)
 - not practical when prototyping
 - usually implemented after a working prototype is available and the statistical method is available for “prime” time
- Parallel computing
 - execute R codes in parallel on multi-core machines or on a cluster of machines



Parallel Computing in R

- Suitable for statistical analyses that are *embarassingly* parallel
 - “little or no effort in separating the problem into a number of parallel tasks” (wikipedia)
 - often the case when there exists no dependency between the parallel tasks
 - simulation studies
 - bootstraps
 - anything that can be phrashed into a `for` loop



Outline

- 1 Introduction
Motivation
- 2 snow and snowfall
Setup
Examples



Requirements

- `snow` (**S**imple **N**etwork **O**f **W**orkstations) is the package that implements parallel computing in R
- `snowfall` is a wrapper package for `snow` that diminishes the learning curve
- a multi-core system
 - standard for recent computers: dual-core, quad-core, or
- cluster of machines with a common mounted disk space (home directory, `~/`)
 - Assumption: Cluster is UNIX-based



Clusters available at UCI

- **BDUC** (`claw1 - claw4`) from UCI's OIT/NACS
 - 1 subcluster of 4 dual-processor dual-core AMD64 Opteron nodes with 32 GB RAM running 64-bit Kubuntu Ibex (8.10) Desktop Edition
 - email `bduc-request@uci.edu` for an account
- **family guy** cluster from Donald Bren School of ICS
 - Dual 2.2GHz Single Core AMD Opteron 248 cpus
 - 4-16GB RAM
 - 64 bit CentOS Linux
 - 20 machines
 - ICS-affiliates to get an account at ICS computer lab



Setup on Cluster

- ssh to `bduc.nacs.uci.edu` or `family-guy.ics.uci.edu`
- passwordless ssh
 - snow launches multiple R instances on different nodes (computer) on the cluster through ssh (if using SOCK mode)

<http://www.oit.uci.edu/computing/bduc/newuser.html>

```
# for no passphrase, use
ssh-keygen -b 1024 -N ""
# if you want to use a passphrase:
ssh-keygen -b 1024 -N "your passphrase"
ssh-copy-id your_bduc_login@bduc.nacs.uci.edu
# you'll have to enter your password one last time
# to get it there.
```



R on BDUC

- Default R on BDUC is broken
- On BDUC, type the following in bash or add to your `.bashrc` file:

```
export  
PATH=/usr/local/R-2.10.0/bin:$PATH
```
- Once logged into the BDUC head cluster, `ssh` into any of the claw nodes: `claw1 - claw4`
 - can't use the head node for parallel computing



snowfall functions

- `sfInit`: initialize the cluster (which machines and number of cpu's)
 - if `passwordless ssh` is not enabled, then you have to type in your password times the number of cpus you request
- Export functions
 - `sfLibrary`: libraries to call on each node
 - `sfSource`: files to source on each node
 - `sfExport`: objects to export to each node
 - `sfExportAll`: export all objects in workspace to each node



snowfall functions, *cont.*

- `sfClusterSetupRNG`: set up random number generation in network
 - `sfClusterSetupRNGstream`: uses the `rlecuyer` package in R - set a random seed on each node
 - `sfClusterSetupSPRNG`: uses the `rsprng` package - set a single seed
- Functions to send parallel code
 - `sfLapply`, `sfSapply`, `sfApply`
 - analogous to `lapply`, `sapply`, `apply`
 - wrap parallel code into a wrapper function
- `sfStop`: stop cluster



Outline

- 1 Introduction
Motivation
- 2 snow and snowfall
Setup
Examples



Example 1: Evaluating OLS for non-Gaussian errors

- In classical linear regression, it is assumed that the response y is related to the predictor x in the following way:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

for $i = 1, \dots, n$ where $\epsilon_i \stackrel{iid}{\sim} \mathcal{N}(0, \sigma^2)$

- In this case, the OLS estimator $\hat{\beta}$, which is also the ML estimator, is known to be

$$\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2 (X^T X)^{-1})$$

- How does the OLS estimator perform (biasedness, consistency, and coverage probability) if we relax the normality assumption?



Simulation Study

- Simulate 10,000 data sets of x and y
- Error will come from $\text{Unif}(-3,3)$ distribution
 - NOTE: Constant variance holds
- $\beta_0 = 1, \beta_1 = 2$
- Record the estimated coefficients and compute 95% confidence intervals under Gaussian errors assumption
- End of day: Take average of coefficients (biased?) and take percentage of times the confidence interval contain the true parameter



Simulation Study, *cont.*

```
## set random seed for reproducibility
set.seed( 12345 )
## wrapper function
Iteration <- function( iter=1, n=100 ){
  x <- rnorm( n, mean=2, sd=2 )
  eps <- runif( n, -3, 3 )
  y <- 1 + 2 * x + eps
  fit <- lm( y ~ x )
  return( cbind( fit$coef, confint( fit ) ) )
}
Iteration()

nsim <- 10000
```



Simulation Study, *cont.*

```
## simlatoon
library( snowfall )
## 1. initialize cluster
sfInit(socketHosts=rep(c('claw1', 'claw2'
                        , 'claw3', 'claw4')
                      ,each=2), cpus=8,type='SOCK'
        ,parallel=T)
## 2. Export necessary stuff: sfLibrary, sfSource, sfEx
sfExportAll( )
## 3. Set up random number generator on network
## sfClusterSetupRNG( ) ## default
sfClusterSetupRNG( type="SPRNG", seed=12345 )
## 4. Execute parallel computations
result <- sfLapply( 1:nsim, Iteration, n=100)
## 5. Stop Clusters
sfStop( )
```



Results of Simulation Study

- With 8 cpu's on the BDOC cluster, the above simulation took less than 8 seconds

```
> length( result )
[1] 10000
> result[1] ## result is a list
[[1]]
                2.5 %    97.5 %
(Intercept) 0.9635419 0.5353903 1.391693
x            1.9499171 1.7982004 2.101634

> coef.matrix <- sapply( result, function( x ) x[, 1] )
> dim( coef.matrix )
[1]      2 10000
```



Results of Simulation Study, *cont.*

```
> ## mean of estimated coefficients
> rowSums( coef.matrix, na.rm=TRUE ) / nsim
(Intercept)          x
  0.999891      2.000276
> ## coverage for intercept
> sum( sapply( result, function( x ){
+   x[1, 2] < 1 & x[1, 3] > 1 } )
+     , na.rm=TRUE ) / nsim
[1] 0.95
> ## coverage for slope
> sum( sapply( result, function( x ){
+   x[2, 2] < 2 & x[2, 3] > 2 } )
+     , na.rm=TRUE ) / nsim
[1] 0.9521
```



Implications

- OLS estimator seems to be unbiased
- Confidence intervals seem to attain the proper coverage probability
- This is all with $n = 100$
- Increase n to investigate consistency
- Try other distributions with a constant variance for the error term
- Decrease n to see when the estimator breaks
- Let the error have non-constant variance (depend on x) and see what happens



Example 2: Bootstrap pairs in Linear Regression

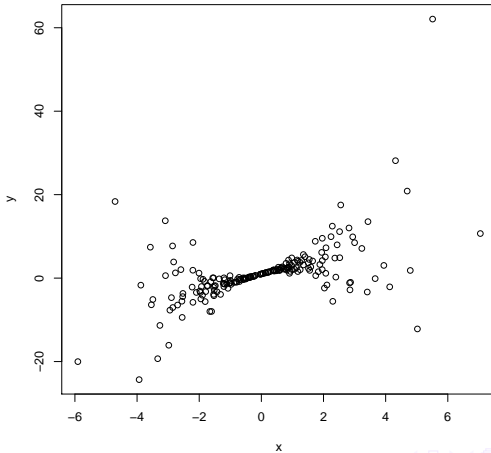
- Let's consider the case of non-constant variance in linear regression

$$\epsilon_i \stackrel{i.i.d.}{\sim} \mathcal{N}(0, x_i^4)$$

- Bootstrap pairs of (x, y) to compute the standard error of the regression coefficients
 - allow for a mean-variance relationship
- Compare this with the model-based standard errors



Scatterplot





Bootstrap Simulation

```
set.seed( 987654321 )  
n <- 200  
x <- rnorm( n, sd=2 )  
eps <- rnorm( n, sd=x^2 )  
y <- 1 + 2*x + eps  
##plot( x, y )  
fit <- lm( y ~ x )  
summary( fit )  
nsim <- 10000
```



Bootstrap Simulation, *cont.*

```
bsPairs <- function( iter=1 )  
{  
  s <- sample( 1:n, replace=TRUE )  
  fit <- lm( y ~ x, subset=s )  
  fit$coef  
}  
bsPairs( )
```



Bootstrap Simulation, *cont.*

```
library( snowfall )
sfInit( socketHosts=rep( c(
  'rupert.ics.uci.edu'
  , 'joe-swanson.ics.uci.edu'
  , 'glenn-quagmire.ics.uci.edu'
  , 'diane-simmons.ics.uci.edu'
  , 'cleveland-brown.ics.uci.edu'
  , 'chris-griffin.ics.uci.edu'
  , 'brian-griffin.ics.uci.edu'
) , each=2) , cpus=14 , type='SOCK' , parallel=T)
sfExportAll( )
sfClusterSetupRNG( )
result <- sfLapply( 1:nsim , bsPairs)
sfStop( )
```



Bootstrap Simulation Results

On 14 cpu's on the family guy cluster, it took us less than 4 seconds

```
> result.matrix <- sapply( result, function( x ) x )
> dim( result.matrix )
[1]      2 10000
> apply( result.matrix, 1, sd ) ##bootstrapped SE
(Intercept)          x
 0.4343742    0.4630120
> summary(fit)$coef
              Estimate Std. Error  t value      Pr(>|t|)
(Intercept)  1.132245   0.4376099  2.587338 1.038822e-02
x            1.882814   0.2134717  8.819972 5.953942e-16
```

NOTE the difference in standard error estimate for x



Comments

- if your workstation is multi-core and you want to use it instead of a cluster, use `localhost` as the node name
- in BDUC and family-guy, find the name of nodes in `/etc/hosts`
- don't request more cpus than a node has
 - the R instances will just be sharing the cpu then
- pass back only the needed data
 - large amount can slow things down, which defeats our purpose
- Be mindful of the shared resources
 - don't hog!
 - use `top` and `ps aux` to see if people are using the machines



Comments, *cont.*

- Ideally, we would want to use `snow` and `R` with the Sun Grid Engine (SGE) for it to allocate resource
 - this isn't set up yet
- I've outlined `snow` using the `SOCK` mode, which is probably the easiest but least robust method
 - Other methods include MPI, PVM, or NWS
- Use `screen` if your job takes a while
- Using the techniques presented in this talk, I reduced jobs that usually take me 2 days to 2 hours!



Comments (Luke Tierney)

- Communication is much slower than computation.
- Communication is serialized in this simple design.
- Some R functions produce large results—reduce on the compute node before returning.
- The PVM and MPI versions are more robust in terms of making sure no stray jobs are left:
 - Using the PVM console or lamhalt and lamwipe you can kill all R processes by halting the PVM or LAM-MPI universe.
 - The socket implementation should not leave any nodes running when the master exits, but it could in some odd circumstances.
 - Occasionally things can go wrong and manual cleanup is needed.



References I

- 1 Knaus, J., Porzelius, C., Binder, H. and Schwarzer, G. (2009). Easier Parallel Computing in R with snowfall and sfCluster. *The R Journal* **1**, 54-59.
- 2 <http://www.oit.uci.edu/computing/bduc/newuser.html>
- 3 <http://www.imbi.uni-freiburg.de/parallel/>
- 4 <http://cran.r-project.org/web/packages/snowfall/vignettes/snowfall.pdf>
- 5 <http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html>
- 6 <http://www.sfu.ca/~sblay/R/snow.html>