Login to Linux server on Windows machine

Active the X Window System display server

Click the ”Start” on lower left in search box: [Xming] [return]

Click on Xming 6.9a

When an icon of ”X” shape appears on lower right corner, the Xming is actived.

Xming provides the visualization ability on local machine which is connected to a remote server.

Login to Linux server on Windows machine

Active the connection between local machine and server

Click the Start on lower left in search box: [SecureCRT] [return]

Click on SecureCRT 6.7.4

Click ”File” on the top right corner, then click ”Connect in Tab”. A new window named Connect in Tab will pop out.

Click the third icon on the top of ”Connect in Tab” window, which is ”New Session”.

In the ”New Session Wizard” window, for SecureCRT protocal, choose SSH2. Then click Next.

For Hostname, type mimosa.stat.purdue.edu, keep the Port as 22, Firewall as None, Username type your Purdue account. Then click Next.

For SecureFX protocol, choose SFTP, then click Next.

Session name and Description can be typed in whatever you want, or keep them as default, and then click Finish.

Login to Linux server on Windows machine

Back to the ”Connect in Tab” window, right click on the mimosa.stat.purdue.edu, and then click Properties.

In the ”Session Options” window, on right hand side, click on Remote/X11.

Check the box of Forward X11 packets, then click OK.

Then click Connect.

Type in your Username and Passward which are your Purdue account and the passward for that account. Then click OK.

Now you are successfully connecting to the mimosa server which is running a Linux operating system.

All Linux command are available to be used, such as ls, pwd, mkdir.

Create folder in Linux system

First create a directory to be the custom location of R packages.

```
mkdir library
```

Now you can see that you create a folder named library by using command:

```
ls
```

Then you can start a R instanse by typing:

```
R
```

Default Location of Packages

R function .libPaths() gets and sets the search path of R packages

Call .libPaths() with no arguments shows the current search path

```
> .libPaths()
```

By default, R installs packages to the first element of .libPaths()

When load packages, R searches in all elements of .libPaths()

Install SNOW package

Add your custom location to the search path

```
> .libPaths(c("~/library",.libPaths()))
```

Now packages are installed to your custom location by default

```
> install.packages("snow")
```

And packages are searched and loaded from your custom location

```
> library(snow)
```

Mocking up a work function

The pause function will print the seed of random number generation, and then pause the system for seconds, finally output the mean of random numbers.

```
> pause = function(args){
        cat("*", args$seed, "*","\n")
        Sys.sleep(args$seconds)
        set.seed(args$seed)
        return(mean(runif(10)))
}
```

`seed` is the element in `args` which set up the seed of random number generation.

`seconds` is another element in `args` which set up how many seconds the system is going to be pasued.

Mocking up some input arguments

```
> seconds = rep(c(0,3), 5)

> seconds

#Total pause time

> sum(seconds)

> args = lapply(seq_along(seconds), function(i)
list(seed = i, seconds=seconds[i]))

> str(args)
```

`args` is a list with 10 elements, each element is also a list with two elements named `seed` and `seconds`.

`seed` are from 1 to 10, `seconds` are 0, 3, 0, 3, back and forth.

Commonly work in serial

```
system.time({result.1 = lapply(args, pause)})
```

```
result.1 = unlist(result.1)
```

Even though you may not write the serial lapply note this:

If you can’t think of your work as an lapply you can’t use SNOW.

So don’t bother to continue...

Simplest linux parallel computing

`mclapply` is a parallelized version of lapply.

`mc.cores` argument is used to specify the number of cores to use, i.e. at most how many child processes will be run simultaneously.

```
>library(parallel)

>system.time({result.2 = mclapply(args, pause,
mc.cores=8)})

>result.2 = unlist(result.2)

>result.1 == result.2
```

What if we have more than one machine? SNOW

```
>library(snow)
```

First specify how many cores want to be occupied,

```
>workers <- rep("mimosa", 2)

>workers

>cl <- makeSOCKcluster(workers)

>snow.time(clusterApplyLB(cl, args, pause))

>cl

>clusterCall(cl, sessionInfo)

>clusterCall(cl, Sys.info)
```

Three types of cluster applies

```
>t3 <- snow.time({result.3 = clusterApply(cl, args,
pause)})

>t4 <- snow.time({result.4 = clusterApplyLB(cl,
args, pause)})

>t5 <- snow.time({result.5 = parLapply(cl, args,
pause)})

>t3

>t4

>t5
```

Three types of cluster applies

ParLapply is not as good as clusterApplyLB in general. It was just a coincidence that it had a good time. Consider:

```
>seconds <- rep(c(0,3), each = 5)

>args <- lapply(seq_along(seconds), function(i)
list(seed = i, seconds=seconds[i]))

>t6 <- snow.time({result.6 = clusterApply(cl, args,
pause)})

>t7 <- snow.time({result.7 = clusterApplyLB(cl,
args, pause)})

>t8 <- snow.time({result.8 = parLapply(cl, args,
pause)})
```

The results from `parLapply` and `clusterApply` depends on the order of the input. In general, `clusterApplyLB` is better than the other two.

clusterCall

`clusterCall` is calling a function on all nodes with same input arguement. Arguments to clusterCall are evaluated on the master, their values transmitted to the worker nodes which execute the function call.

```
>clusterCall(cl, exp, 1)
```

```
>clusterCall(cl, runif, 3)
```

```
>my_func <- function(x) {Sys.sleep(x); cat("Done
with ", x, "\n");x}
```

```
>clusterCall(cl, my_func, 3)
```

clusterExport and clusterEvalQ

`clusterEvalQ(cl, expr),'expr'` is treated on the master as a character string. The expression is evaluated on the worker nodes.

```
>clusterEvalQ(cl, library(lattice))
```

```
>clusterEvalQ(cl, runif(3))
```

How about

```
>clusterEvalQ(cl, my_func(3))
```

Why it does not work?

clusterExport and clusterEvalQ

```
>ls()

>clusterEvalQ(cl, ls())

>x <- 1

>clusterExport(cl, "x")
```

To the global environments of each worker node

```
>clusterEvalQ(cl, ls())

>clusterExport(cl, "my_func")

>clusterEvalQ(cl, my_func(3))
```

parApply

```
parApply(cl, X, MARGIN, fun, ...)
```

```
>A <- matrix(1:10, 5, 2)
```

```
>parApply(cl, A, 1, sum)
```

’1’ indicates rows, ’2’ indicates columns, ’c(1,2)’ indicates rows and columns.

```
>parCapply(cl, A, sum)
```

```
>parRapply(cl, A, sum)
```

```
>clusterSplit(cl, 1:10)
```

parallel matrix computation

```
>A<-matrix(rnorm(10000),100)

>t9 <- snow.time(A %*% A)

>t10 <- snow.time(parMM(cl,A , A))
```

Computation time for parallel is longer due to the communication. communication is orders of magnitude slower than computation.

```
>A <- matrix(rnorm(4000000),2000)

>t11 <- snow.time(A %*% A)

>t12 <- snow.time(parMM(cl,A , A))
```

In other words, parallel is more efficient for large and complex data.

Data analysis example

Suppose we want to run a linear regression on a simulated data with 3 variables and 10,000,000.

```
> data_fun <- function(seed,n){
        set.seed(seed)
        x1 <- rnorm(n)
        set.seed(seed-1)
        x2 <- rnorm(n)
        set.seed(seed+1)
        eps <- rnorm(n)
        y = 1.1*x1 + 3.6*x2 + eps
        df <- data.frame(cbind(x1,x2,y))
        return(df)
}

> sim <- data_fun(1,10000000)
```

Data analysis example

The slit the whole dataset to two parts.

```
>subsim <- split(sim, sample(1:2))
```

```
> str(subsim)
```

Run the regression on the whole dataset.

```
> system.time({lm(sim$y ~ sim$x1+sim$x2)})
```

Distribute the two subsets to two workers, and run the regression independently.

```
> snow.time(clusterApplyLB(cl, subsim,
        function(r)lm(r$y ~ r$x1 +r$x2)))
```