Workshop in Methods: Advanced R

Jefferson Davis, Hui Zhang, and Scott Michael
Research Technologies, Visualization and Analytics
Indiana University

December 5, 2014
Workshop Outline

- Motivation: Why use R?
- Using R on central research systems
- Submitting R batch jobs to central systems
- Advanced analysis in R
- Advanced plotting in R
- Introduction to parallel R
Why Use R?

• R is open source which means it is:
  – freely available
  – flexible
  – extensible

• R also offers a variety of parallel paradigms which will allow one to tackle big data problems that are either memory or compute bound with multiple computers in a cluster
IU Central Research Systems: Compute Systems

- Three main computational systems and several storage options
  - Accounts can be created at http://itaccounts.iu.edu
  
  - Karst: high throughput computing (karst.uits.iu.edu)
    - 256 computational nodes (16 cores & 32 GB memory/node)
  
  - Big Red II: high performance computing (bigred2.uits.iu.edu)
    - 1020 computational nodes; 676 GPU based, 344 CPU only
      (16/32 cores & 32/64 GB memory/node)
  
  - Mason: large memory computing (mason.uits.iu.edu)
    - 16 computational nodes (32 cores & 512 GB memory/node)

- All systems are “batch scheduled” and use the PBS scheduler to manage computational jobs
IU Central Research Systems: Access

• You can use R through the command line on any of the central systems
  – Command line tools are available via:
    • Putty or OpenSSH for Windows
    • Terminal or Xquartz on Mac
    • Terminal on Linux

• A centralized R development environment is available at https://rstudio.iu.edu
  – This requires an account on Karst and links to your Karst home directory
IU Central Research Systems: Data Storage

• There are three main storage options for central systems, which can be used separately or in conjunction
  – Home Directories: Each user is allotted 100GB of storage in the home directory filesystem which is the initial directory you start in when logging into the system. This filesystem serves all compute platforms and the 100GB limit is an aggregate across all platforms
  – Data Capacitor 2: High performance Lustre based filesystem with 3.5PB of total storage. Each user gets up to 10TB of storage to begin, which can be expanded upon request. Files that have not been accessed in 90 days are subject to purging.
  – Scholarly Data Archive: 14PB of tape storage intended for long term archiving of data
IU Central Research Systems: Getting started

• Let’s copy some example R scripts that will be used throughout this workshop. The scripts are located on the Data Capacitor at /N/dc2/scratch/scamicha/WIM_AdvancedR

• Command line:
  
  cp -r /N/dc2/scratch/scamicha/WIM_AdvancedR ~

• Rstudio

  Use shell functionality in Rstudio
IU Central Research Systems: Configuring Environment

• **Modules**
  – The modules system controls a user’s software environment on the central systems. It is initialized with a file named `.modules` in your home directory
  – Add R module to `.modules`

• **R package library**
  – R gives access to a ton of user contributed functionality via the package system. Packages need to be installed into a given version of R and then need to be loaded in each session that you want to use them
  – Packages can be installed anywhere but we will focus on two categories: centrally installed and installed locally to a user directory
IU Central Research Systems: Configuring Environment

• Centrally installed libraries
  – R come with a long list of preinstalled packages and packages can be requested to be installed into the R distribution on any of the systems (researchanalytics@iu.edu)

• Locally installed libraries
  – Create a location for library installation
  – Set installation directory in .Renviron file
  – Use install.packages or R CMD INSTALL
IU Central Research Systems: Batch Jobs

- All central systems are scheduled by a batch scheduler to ensure maximum efficiency and throughput of computational workload
- Scheduling is done via PBS/Torque resource allocation is handled by Moab. Priority is set via fairshare
- Several important PBS commands
  - qsub
  - qstat
  - qdel
- Several important Moab commands
  - showstart
  - showbf
  - mdiag -f
IU Central Research Systems: For loop example

- R scripts can be run via Rscript or R CMD BATCH
  - Output and handling of command line variables are slightly different, Rscript is probably better
- Jobs can be submitted and checked via command line or Rstudio shell interface
Advanced Plotting in R
Plotting in 2d

- The most basic plotting command in R is `plot()`
- As a high-level function it will create axes, tick marks, etc.
- Many user-written classes will have default `plot()` functions that act reasonably
Plotting in 2d

- The cars data frame is a two-column data set of cars speeds and stopping distances from the 1920s

```r
head(cars)
speed  dist
1       4     2
2       4     10
3       7      4
4       7     22
5       8     16
6       9     10
```
Plotting in 2d

- By default `plot()` produces a scatterplot
  
  ```r
  plot(cars)
  ```

- Axis labels are from the names in the data frame

- Axis scale is from the range of the data
Plotting in 2d

- You can change the defaults in `plot()`

```r
plot(cars$speed, cars$dist, main = "A Title", xlab = "The Speeds", ylab = "The Distances", col = "steel blue")
```

Switched The Speeds

Switched The Distances

The Distances

The Speeds
Plotting in 2d

- Since `plot()` is a high-level function, each call will by default create its axes, labels, etc.
- Combining `plot()` calls can be messy.

```r
plot(cars,type="p")
par(new=TRUE)
plot((lowess(cars))
```
- The graphs labels are clearly bad. While not obvious, the scales don’t match either.
Plotting in 2d

• To add details it’s better to use low-level functions.
  
  ```r
  plot(cars,type="p")
  line(lowess(cars),col="red")
  ```
Combining plots

- To show multiple plots in one graphics window use the `par()` command with the `mfrow` parameter

```r
par(mfrow=c(2,2))
plot(cars,type="p")
plot(cars,type="l")
plot(cars,type="h")
plot(cars,type="s")
```
Plotting in 2d

- The default plot for a data frame with many columns is a matrix of scatterplots.
  ```r
  head(mtcars, 2)
  mpg cyl disp hp drat wt qsec vs am gear carb
  Mazda RX4  21.0  6  160  110  3.90  2.620 16.46  0  1    4    4
  Mazda RX4 Wag 21.0  6  160  110  3.90  2.875 17.02  0  1    4
  ```
  ```r
  plot(mtcars)
  ```
- This is rarely helpful
Interactive plot functions

- Interactive functions let you interact with your graphics.
- Good for when you are still getting a feel for your data

```r
plot(mtcars$disp, mtcars$mpg)
identify(mtcars$disp, mtcars$mpg, labels=row.names(mtcars))
```
Writing graphics to a file

Good for batch work where you don’t need to see the file in real time

```r
fit<-lm(faithful$waiting ~ faithful $eruptions)
png()
plot(faithful$eruptions , faithful $waiting)
abline(fit$coefficients)
dev.off()
```

Default name is “Rplot001.png”.

![Plot](image-url)
Plotting in 3D

- Some data is well-suited to three dimensional plots
- The matrix volcano records elevations of the volcano Maunga Whau in New Zealand

```
volcano[1:4,1:4]
[1,] 100 100 101 101
[2,] 101 101 102 102
[3,] 102 102 103 103
[4,] 103 103 104 104
```

dim(volcano)
x<-1:87
y<-1:61
Plotting in 3D

#Default are a heat map
image(x, y, volcano)
Plotting in 3D

```python
# Changing the color map
image(x, y, volcano, col=terrain.colors(range(volcano)))
```
Plotting in 3D

#A contour map
contour(x,y,volcano)
Plotting in 3D

#A perspective plot
persp(x,y,volcano)
Plotting in 3D

# Changing the viewing angle
# (Azimuth and coaltitude)
persp(x,y,volcano,theta=25,phi=45)
Plotting in 3D

#An interactive example
#How this behaves may depend
#on your windowing system

library(rgl)
surface3d(x, y, volcano)
The package ggplot2

- The ggplot2 package was created in 2005 by Hadley Wickham
- It implements the object oriented design ideas of Leland Wilkinson’s *The Grammar of Graphics.*
- Wilkinson in turn was strongly influenced by the ideas of cartographer Jacques Bertin
The package ggplot2

The function `qplot()` is a ggplot2 version of the regular function `plot()`

```
library(ggplot2)
qplot(speed, dist, data = cars)
```
The package ggplot2

Combining geoms can produce more complex plots

```r
library(ggplot2)
library(foreign)
dat <- read.dta("http://www.ats.ucla.edu/stat/data/ologit.dta")

ggplot(dat, aes(x = apply, y = gpa)) +
geom_boxplot(size = .75)
```
The package ggplot2

```r
p <- ggplot(dat, aes(x = apply, y = gpa)) + geom_boxplot(size = .75)
p <- p + geom_boxplot(size = .75)
p <- p + geom_jitter(alpha = .5)
p <- p + facet_grid(pared ~ public, margins = TRUE)
p <- p + theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1))
p
```
Parallel Programming in R
The Brutal Truth

- We are here because we love R. Despite our enthusiasm, R has two major limitations, and some people may have a longer list:
  - Regardless of the number of cores on your CPU, R will only use 1 on a default build
  - R reads data into memory by default. The OS and system architecture can only access 4GB physical memory on a 32 bit system, but typically R will throw an exception at 2GB
The Brutal Truth

• And there are a couple of solutions:
  – Build from source and use special build options
  – Use another language like C, FORTRAN, etc
  – Interface R with C and FORTRAN
  – Let external packages and parallelization solve these problems

We will discuss #4 above
Parallelism means running several computations at the same time and taking advantage of multiple cores or CPUs on a single system, or CPU on other system.

R has several packages for parallelism:
- *Rmpi*
  - allows communication and synchronization between processes
- *snowfall*, a non-technical wrapper of *snow*
  - embarrassingly parallel
- Foreach --- by Revolution Computing Inc.
- *Multicore*
  - multiple tasks by multi-cores on one single compute node
- *R interfacing with GPGPU/MIC*
- *PBDR*
  - programming with big data using R
- *Rhadoop*
  - R wrapper to use hadoop
Parallelism means running several computations at the same time and taking advantage of multiple cores or CPUs on a single system, or CPU on other system.

R has several packages for parallelism:
- **Rmpi**
  - allows communication and synchronization between processes
- **snowfall**, a non-technical wrapper of **snow**
  - **embarrassingly parallel**
- **Foreach** --- by Revolution Computing Inc.
- **Multicore**
  - multiple tasks by multi-cores on one single compute node
- **R** interfacing with **GPGPU/MIC**
- **PBDR**
  - programming with big data using **R**
- **Rhadoop**
  - **R** wrapper to use hadoop
Some Basic Terminologies

- A **CPU** is the main processing unit in a system. Today most systems have one processor with between one and four cores. Some have two processors, each with one or more cores.
- A **cluster** is a set of machines that are all interconnected and share resources as if they were one giant computer.

<table>
<thead>
<tr>
<th>System summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine type</td>
<td>Hybrid (x86_64 CPUs/NVIDIA Kepler GPUs)</td>
</tr>
<tr>
<td>Operating system</td>
<td>Cray Linux Environment (based on SUSE Linux SLES 11)</td>
</tr>
<tr>
<td>Memory model</td>
<td>Distributed</td>
</tr>
<tr>
<td>Nodes</td>
<td>1.020</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computational system details</th>
<th>Total</th>
<th>Per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>1.364</td>
<td>2 (compute) 1 (GPU)</td>
</tr>
<tr>
<td>Processor cores</td>
<td>21.824</td>
<td>32 (compute) 16 (GPU)</td>
</tr>
<tr>
<td>RAM</td>
<td>43,648 GB</td>
<td>64 GB (compute)</td>
</tr>
</tbody>
</table>
Some Basic Terminologies

- **A CPU** is the main processing unit in a system. Today most systems have one processor with between one and four cores. Some have two processors, each with one or more cores.
- **A cluster** is a set of machines that are all interconnected and share resources as if they were one giant computer.

<table>
<thead>
<tr>
<th>System summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine type: Hybrid (x86_64 CPUs/NVIDIA Kepler GPUs)</td>
</tr>
<tr>
<td>Operating system: Cray Linux Environment (based on SUSE Linux SLES 11)</td>
</tr>
<tr>
<td>Memory model: Distributed</td>
</tr>
<tr>
<td>Nodes:</td>
</tr>
<tr>
<td>Computational system details</td>
</tr>
<tr>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>CPUs:</td>
</tr>
<tr>
<td>Processor cores:</td>
</tr>
<tr>
<td>RAM:</td>
</tr>
</tbody>
</table>
Some Basic Terminologies

- **master/slave model in parallelization**
  - Start with an “embarrassingly parallel” problem

![Single process diagram]

- Divide jobs among slave processes and collect results

![Master and slaves diagram]

- Ideal: $p$ times faster with $p$ slaves
SnowFall package

1. Initialize with sfInit.
2. Load the data and prepare the data for parallel calculation
3. Wrap computation code into a function that can be called by an R list function
4. Export objects needed in the calculation to cluster nodes and load the necessary packages on them
5. Start a network random number generator to ensure that each node produces random number independently (optional)
6. Distribute the calculation to the cluster using a parallel list function
7. Combine and compile results from slaves
8. Stop the cluster
A toy example to improve using SnowFall

```r
myfunction <- function(arg1) {
  retv <- 1
  for (i in 1:arg1)
    retv <- (retv * i)
}

system.time(lapply(1:3000, myfunction))
```

<table>
<thead>
<tr>
<th>User</th>
<th>System</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.189</td>
<td>0.001</td>
<td>1.189</td>
</tr>
</tbody>
</table>
A toy example to improve using SnowFall

- library(snowfall)
- #parse node list
  con <- file("nodes.cfg", "rt")
  nodes <- vector()
  print("read in nodelist4.cfg")
  while(length(input <- readLines(con, n = 1000)) > 0) {
    for (i in 1 : length(input)){
      print(tolower(input[i]))
      nodes <- append(nodes, tolower(input[i]))
    }
  }
  sfInit(parallel=TRUE, cpus=32, type="SOCK", socketHosts=nodes)

- myfunction <- function(arg1) {
  retv <- 1
  for (i in 1:arg1)
    retv <- (retv * i)
}
- sfClusterSetupRNG()
- # Start the clock!
  ptm <- proc.time()
  result <- sfLapply(1:3000, myfunction)
  # Stop the clock
  proc.time() - ptm
  sfStop()
Multicore package (now part of parallel package on Karst)

• **multicore** provides functions for parallel execution of R code on systems with multiple cores. **multicore jobs all share the same state when spawned.**

• The main functions in multicore are:
  - `mclapply`, a parallel version of `lapply`
  - `parallel`, do something in a separate process
  - `collect`, get the results from calls to parallel
Multicore package

- **mcapply** is a parallel version of **lapply**. Works similar to lapply, but has some extra parameters:

  - **mcapply**(*X*, **FUN**, ..., **mc.preschedule** = **TRUE**, **mc.set.seed** = **TRUE**, **mc.silent** = **FALSE**, **mc.cores** = `getOption("cores")`)

  **mc.cores** controls the maximum number of processes to spawn
  **mc.preschedule** = **TRUE** controls how data are allocated to jobs
  **TIPS:**
  - **TRUE**: many data in *X*, light computation for each data set
  - **FALSE**: few data in *X*, heavy computation for each data set

  **mc.silent** = **TRUE** suppresses standard output
A toy example to improve using Multicore

```r
• fib <- function (n) {
  • if(n<1)
  •   stop("Input must be an integer >=1")
  • if(n==1|n==2)
  •   1
  • else
  •   fib(n-1)+fib(n-2)
  • }
• # Start the clock!
• ptm <- proc.time()
• sapply(1:35, fib)
• proc.time() - ptm
```
A toy example to improve using Multicore

- `library(multicore)`
- `fib <- function (n) {
  if(n<1)
    stop("Input must be an integer >=1")
  if(n==1|n==2)
    1
  else
    fib(n-1)+fib(n-2)
}
- `ptm <- proc.time()`
- `mclapply(1:35, fib, mc.preschedule=TRUE, mc.cores=16)`
- `proc.time() - ptm`
A toy example to improve using Multicore

library(parallel)
cores <- detectCores()
print(cores)

system.time(lapply(1:3000, rnorm))

Distributing 3000
operations to 16 cores?
You don’t get linear
speedups, due to data
transfer, system level
communications, etc ......

system.time(mclapply(1:3000, rnorm, mc.cores=cores))
library(parallel)
cores <- detectCores()
print(cores)

myfunction <- function(arg1) {
  retv <- 1
  for (i in 1:arg1)
    retv <- (retv * i)
}

system.time(lapply(1:3000, myfunction))

system.time(mclapply(1:3000, myfunction, mc.cores=cores))

Now another toy example to scale better!

Distributing 3000 factorial operations to 16 cores? You get speedups now.
Is parallelization always good?

- library(parallel)
- cores <- detectCores()
- print(cores)
- system.time(lapply(1:16, rnorm))
- system.time(mclapply(1:16, rnorm, mc.cores=cores))

Distributing 16 operations to 16 cores? It slows down due to data transfer, system level communications, etc .......
Questions?

researchanalytics@iu.edu