R:

HPC QUICK GUIDE

REV NUMBER: 1.0
REV DATE: 10/26/2017

TABLE OF CONTENTS

I. WHAT IS R

INTRODUCTION TO R

R is a language and environment for **statistical computing and graphics**. R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, etc.) and graphical techniques, and is highly extensible.

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. [1]

THE R ENVIRONMENT

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it provides:

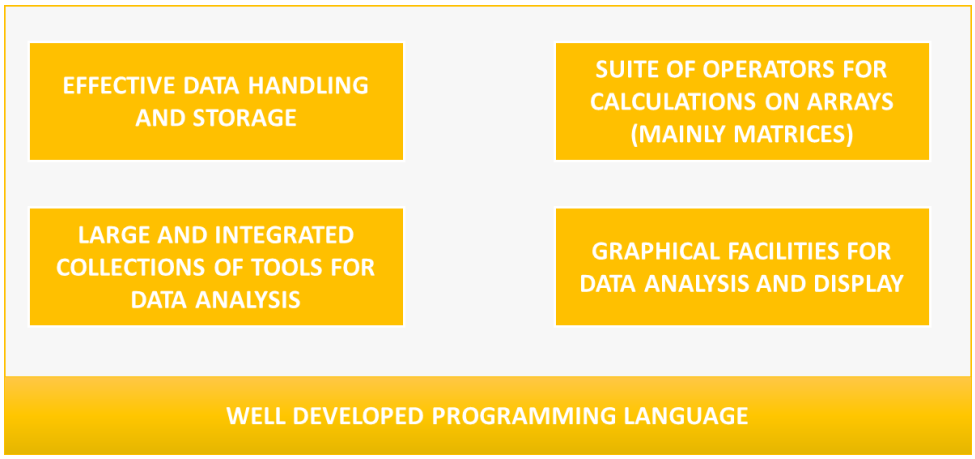

Figure 1: R Environment Representation

R is an environment within which many classical and modern statistical techniques have been implemented. A few of these are built into the base R environment, **but many are supplied as packages**.

There are about 25 packages supplied with R (called *"standard"* and *"recommended"* packages) and many more are available through the CRAN (Comprehensive R Archive Network) family of Internet sites (via https://CRAN.R-project.org) and elsewhere.

See section 4 for details on handling R packages on HPC.

II. QUICK SET-UP GUIDE

CONNECTING TO THE HPC NETWORK

In order to log into the HPC network and begin working with R, you need to ensure the following requirements are met:

- Secure Network: Connect to USC's *Secure Wireless network/Ethernet* or a *VPN Client*.
- Secure shell (Linux CLI):
  - On **Mac**, you may use its native Terminal
  - On **Windows**, you may use your preferred client (i.e. PuTTY, X-Win32, MobaXTerm, etc.)

In order to connect to the network, you must log in to one of the login head nodes: **hpc-login2** or **hpc-login3**, ensuring X11 forwarding is enabled (provides the basic framework for a GUI environment):

- **Mac Terminal** → type: "*ssh –X <me>@hpc-login2.usc.edu*" or "*ssh –X <me>@hpc-login3.usc.edu*"
- **Windows** → Set the ssh hostname to **hpc-login2** or **hpc-login3** and launch.

HPC FILE SYSTEM

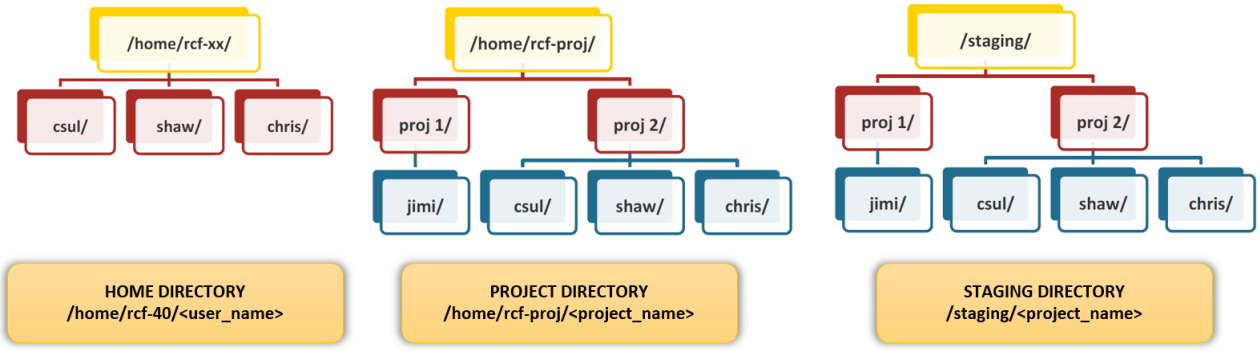

Figure 2: HPC File System Diagram

SOURCING R

The first step to ensure you are using the version of R you intend to use is **sourcing**. HPC maintains installed software in **/usr/usc/**. You may view all available R versions within its subfolder by using the following command:

INPUT:

```
$ ls –l /usr/usc/R
```

OUTPUT:

```
drwxr-xr-x 5 daemon daemon 4096 Sep 28  2016 2.13.0/
drwxr-xr-x 5 daemon daemon 4096 Sep 28  2016 2.14.1/
drwxr-xr-x 5 root   root   4096 Sep 28  2016 2.15.3/
drwxr-xr-x 5 root   root   4096 Sep 28  2016 3.0.2/
drwxrwxr-x 5 root   root   4096 Sep 28  2016 3.1.1/
drwxr-xr-x 5 daemon daemon 4096 Sep 28  2016 3.2.1/
drwxr-xr-x 5 root   root   4096 Oct 21  2016 3.3.1/
drwxr-xr-x 5 root   root   4096 Feb  8  2017 3.3.2/
drwxr-xr-x 5 root   root   4096 Jun 30 14:06 3.4.0/
lrwxrwxrwx 1 root   root      5 Oct 20  2016 default -> 3.3.1/
```

Figure 3: R Available Versions under /usr/usc/

NOTE: "**default**" is a symbolic link (soft link), which means it is a special kind of file that points to another file (shortcut). HPC only updates the symbolic link to the most recent versions of available software once a year, during downtime, which means new software may be available but not linked to be the default. **Please ensure you source the desired version directly.**

Within each version directory there are **two setup scripts**:

- "setup.sh" (for use with the bash shell, which is the default)
- "setup.csh" (for use with the "t" or "c" shells)

In order to source the version you want, you must type in the following:

INPUT:

```
$ source /usr/usc/R/<desired_version>/setup.sh
```

Where *<desired_version>* can be any of the available versions within /usr/usc/ such as 3.4.0, **3.3.1**, etc.

```
hachuelb@hpc-login3:/usr/usc$ source /usr/usc/R/3.3.1/setup.sh
hachuelb@hpc-login3:/usr/usc$ R

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

Figure 4: Example for sourcing the default version of R

NOTE: Please ensure you **always source the version you require** for your particular project, as the default version that ships with the Operating System is not maintained by HPC and **may not be available on the compute nodes**. Sourcing the version of your choice will guarantee reproducibility and is a best practice.

RUNNING R

After you've sourced the version of R of your choice (when working on the command line), the **'R' command** can be used to start the main R program.

INPUT:

```
$ R
```

OUTPUT:

```
hachuelb@hpc-login3:/usr/usc$ R
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

Figure 5: R Initialization Output Screenshot

At this point, **R commands may be issued.**

As you may have noticed at the top of the output results, you can see the **version of the R program** that has been initialized (i.e., 3.3.1). This is the currently sourced R version.

III. SOME QUICK R EXAMPLES

SAMPLE R INTERACTIVE SESSION

If you consider yourself a novice in R, you may want to go through the following interactive session. This should give you some familiarity with the style of R sessions and more importantly some instant feedback on what actually happens.

a) Start R by using the 'R' Command

```
$ R
```

The R program begins, with a banner.

b) Generate two pseudo-random normal vectors of x- and y-coordinates

```
$ x <- rnorm(50)
$ y <- rnorm(x)
```

c) Plot the points in the plane. A graphics window will appear automatically

```
$ plot(x, y)
```

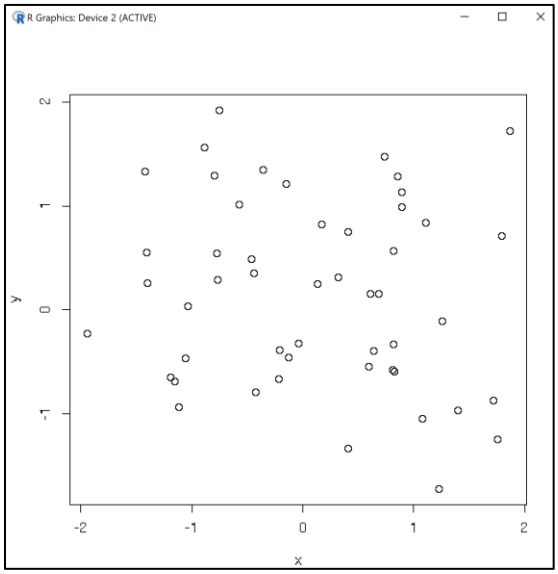

Figure 6: Sample graphics window in R

NOTE: To ensure you are able to see the R graphics displayed in a window (as shown above), **X11 forwarding must be enabled**. Please refer to section 2 (connecting to the HPC network) for further details.

If you wish to save the generated plot: In interactive mode, at startup time, R initiates a graphics device driver which opens a special graphics window for the display of interactive graphics (done automatically, with X11 forwarding).

1. Choose the format you want to use (i.e. JPG file)
2. The only argument that the device drivers need is the name of the file that you will use to save your graph. **Remember that your plot will be stored relative to the current directory**:

3. So if I wanted to save a jpg file called "rplot.jpg" containing a plot of x and y, you would type the following commands:

```
$ jpeg('rplot.jpg')
$ plot(x, y)
$ dev.off()
```

Note that if you do not specify a path, the plot will be saved in the directory R was started in.

For more information on graphical procedures with R: https://cran.r-project.org/doc/manuals/R-intro.html#Graphics/

d) Check which objects are currently in your workspace

```
$ ls()
```

The output will show the objects "x" and "y".

e) Remove the objects from the workspace (clean up)

```
$ rm(x, y)
```

f) Create an array x = (1, 2, …, 20), a weight vector w of standard deviations, and make a data frame of two columns x and y. Type the name of the data frame object to view it.

```
$ x <- 1:20
$ w <- 1 + sqrt(x)/2
$ dummy <- data.frame(x = x, y = x + rnorm(x) * w)
$ dummy
```

```
    x         y
1   1  0.6871524
2   2  4.7411920
3   3  1.9550376
4   4  5.7902086
5   5  8.5176842
6   6  9.9005728
7   7  7.9861938
8   8  8.8800371
9   9 10.7172413
10 10 17.4480119
11 11 14.9751005
12 12 11.8843626
13 13 16.1579211
14 14 13.9614361
15 15 17.5861778
16 16 10.9985661
17 17 16.8372898
18 18 21.0685752
19 19 19.6977936
20 20 17.2114784
```

Figure 7: dummy data frame output

g) Fit a simple linear regression and look at the analysis. With y to the left of the tilde (~), we are modelling y dependent on x

```
$ fm <- lm( y ~ x, data = dummy)
$ summary(fm)
```

```
Call:
lm(formula = y ~ x, data = dummy)

Residuals:
    Min      1Q  Median      3Q     Max
-5.7074 -1.0566 -0.0648  1.8282  6.0394

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.5798     1.2450   2.072   0.0529 .
x             0.8829     0.1039   8.495 1.03e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.68 on 18 degrees of freedom
Multiple R-squared:  0.8004,    Adjusted R-squared:  0.7893
F-statistic: 72.17 on 1 and 18 DF,  p-value: 1.031e-07
```

Figure 8: Summary of linear regression output sample

h) Quit the R program

```
$ q()
```

Once you quit, you will be asked if you want to **save the R workspace.** The workspace is your current R working environment and includes any user-defined objects. At the end of an R session, you can save an image of the current workspace and it will be automatically reloaded the next time R is started. Workspaces are **.RData** files.

To save a workspace image, you must run R from your home directory. If you have previously saved a workspace image and wish to prevent R from restoring the image, **you must delete (rm) the .RData file**, which is a hidden file saved in your home directory.

For a longer interactive session, please refer to the following link: https://cran.r-project.org/doc/manuals/R-intro.html#A-sample-session/

SAMPLE R IN BATCH MODE

NOTE: R can be run on the cluster either interactively or in batch mode. In either case, you must use the scheduler to allocate yourself a compute node, rather than running R on the login node. For the following simple examples, however, you may use the login node for added simplicity.

Let's being by creating a simple "Hello World" R script example. You may use any editor of your choice. Copy the following code into your file and save it as "**myScript.R**" within your current directory.

```
helloWorld <- function(){
        print("Hello World!")
}
helloWorld()
```

When running R scripts (i.e. myScript.R) you can use one of the following options:

a) Executing R from the R console: source()

```
> source("myScript.R")
```

You may run an R script by using the **source()** command, and it may be executed at any time in an R session.

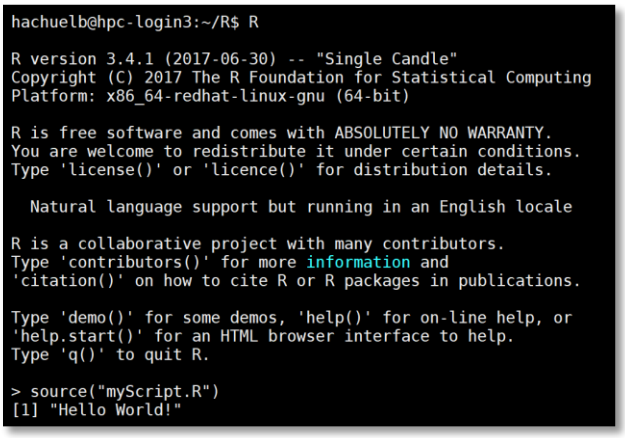

Figure 9: Executing an R script from within an R session

b) Executing R programs from the command line: Rscript

```
$ Rscript myScript.R
```

Notice the script has been saved in a directory named R. This is actually a symbolic link pointing to a directory within a project folder. *Section 4 below (R Packages) goes over the details of creating the link for your own projects.*

Figure 10: Executing an R script through Rscript

Rscript prints its output to stdout. It is currently the preferred method for program execution.

c) Executing R programs from the command line: R CMD BATCH

```
$ R CMD BATCH [options] myScript.R [outfile]
```

With 'R CMD BATCH' you will not see the output of the script on the terminal/GUI, as it is saved in an output file. **The output file lists the commands from the script file and their outputs**. If no outfile is specified, the name used is that of infile and **.Rout** is appended to outfile.

```
hachuelb@hpc-login3:~/R$ ls -la
total 24
drwxr-sr-x 3 hachuelb lc_hpcc 4096 Oct 13 15:32 ./
drwxr-s--- 3 hachuelb lc_hpcc 4096 Sep 29 12:04 ../
drwxr-sr-x 4 hachuelb lc_hpcc 4096 Oct  6 13:40 data/
-rw-r--r-- 1 hachuelb lc_hpcc   63 Oct 13 15:09 myScript.R
-rw-r--r-- 1 hachuelb lc_hpcc  856 Oct 13 15:32 myScript.Rout
```

Figure 11: Snapshot of directory with output file myScript.Rout

```
hachuelb@hpc-login3:~/R$ R CMD BATCH myScript.R
hachuelb@hpc-login3:~/R$ more myScript.Rout

R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
>
> sayHello <- function(){
+       print('Hello World!')
+ }
>
> sayHello()
[1] "Hello World!"
>
> proc.time()
   user  system elapsed
  0.221   0.029   0.242
```

Figure 12: Executing an R script through R CMD BATCH & Reading Output File

IV. R PACKAGES

NOTE: Software and package installations can only be performed through the head nodes (including the data transfer nodes), **as they are the only nodes that can access the public internet.**

At HPC, software can be installed **globally** (*system-wide, located in places such* **/usr/usc/** and require root privileges) or **locally** (into "local" folders)**.**

HPC users, presumably such as yourself, will only perform local or "user" installs into project directories (*see Figure 2 for the HPC file structure diagram*).

R PACKAGES - INTRODUCTION

All R functions and datasets are stored in *packages*. The standard (or base) packages are considered part of the R source code. They contain the basic functions that allow R to work, and the datasets and standard statistical and graphical functions that are described in this manual. They should be automatically available in any R installation. [1]

R contains most arithmetic functions like mean, median, sum, prod, sqrt, length, log, etc.

An extensive list of R functions can be found on:
https://cran.r-project.org/doc/manuals/R-intro.html#Function%20and%20variable%20index/

SETTING UP YOUR DIRECTORY

As mentioned in the note above, users are not allowed to install software, which includes R packages, at a global level. Nevertheless, R makes it very easy for users to install libraries in their home directories.

R's default installation location is ${HOME}/R. However, your **disk quota** on your home directory is 1GB and your **file quota** is 100,000, which can easily be exceeded if downloading packages.

NOTE: If you wish to get information such as memory used and available on your disk and file quota, you may type in the following command on the command line:

```
$ myquota
```

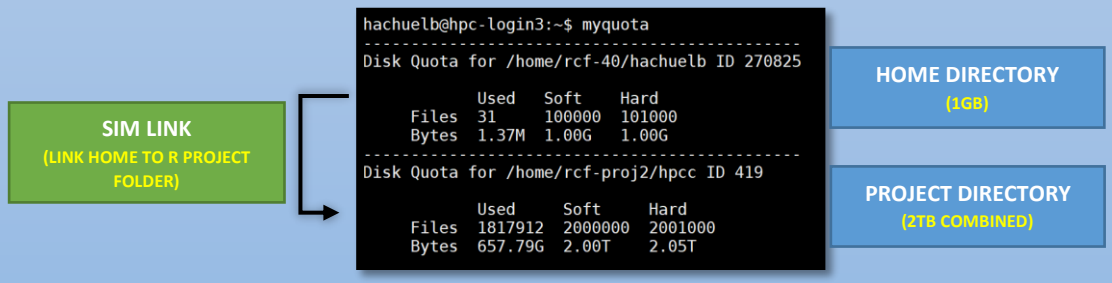

Figure 13: myquota example

Thus, Before you let R install packages in your home directory, and potentially fill the disk space in home, create a new directory for R packages in your project area and symbolically link it to $HOME/R:

```
$ cd /path/to/your/project/directory  #move into your project directory
$ mkdir RPackages  # create a directory - you may name it whatever fits your needs
$ cd ~  # go back to your home directory
$ ln –s /path/to/your/project/directory/RPackages R # create a sim link
```

If you have followed the above steps correctly, you have successfully created a symbolic link between your home directory and your new project directory for R packages. To ensure the link has been successfully created:

INPUT (from your home directory):

```
$ ls –l R
```

OUTPUT:

```
hachuelb@hpc-login3:~$ ls -l R
lrwxrwxrwx 1 hachuelb its-ar 40 Sep 21 14:56 R -> /home/rcf-proj/hpcc/hachuelb/myRPackages/
```

Figure 13: R Symbolic Link to R Packages Project Directory

At this point, files written to ~/R will really be stored in your project (R Packages) directory.

INSTALLING PACKAGES

Before we begin, please ensure you have sourced the version of R you wish to work with. Please refer to *section 2 (Quick Set-Up Guide)* for details on how to do so. For the following examples, the sourced version will be 3.3.1.

A package is loaded from a library by the function **library()**. Thus a library is a directory containing installed packages; the main library is R_HOME/library, but others can be used, for example by setting the environment variable R_LIBS or using the R function .libPaths(). [2]

To see which packages are installed at HPC:

INPUT *(after instantiating an R session):*

```
> library()
```

OUTPUT:

```
Packages in library '/auto/usc/R/3.3.1/lib64/R/library':

base                    The R Base Package
BayesBridge             Bridge Regression
boot                    Bootstrap Functions (Originally by Angelo Canty
                        for S)
car                     Companion to Applied Regression
class                   Functions for Classification
cluster                 "Finding Groups in Data": Cluster Analysis
                        Extended Rousseeuw et al.
coda                    Output Analysis and Diagnostics for MCMC
codetools               Code Analysis Tools for R
compiler                The R Compiler Package
datasets                The R Datasets Package
DEoptimR                Differential Evolution Optimization in Pure R
ergm                    Fit, Simulate and Diagnose Exponential-Family
                        Models for Networks
foreign                 Read Data Stored by Minitab, S, SAS, SPSS,
                        Stata, Systat, Weka, dBase, ...
graphics                The R Graphics Package
grDevices               The R Graphics Devices and Support for Colours
                        and Fonts
```

Figure 14: Output snapshot of R library() command

Within R, using the *install.packages()* function always attempts to install the latest version of the requested package available on CRAN:

```
> install.packages("<package_name>")
# Alternatively, you may install multiple packages at once:
> install.packages(c("<package_name_1>", "<package_name_2>"))
```

If the <package_name> package above depends upon other packages that are not already installed locally, the R installer automatically downloads and installs those required packages. This is a huge benefit that frees users from the task of identifying and resolving those dependencies. [3]

You can also install R from the shell command line. This is useful for some packages when an internet connection is not available or for installing packages not uploaded to CRAN. To install packages this way, first locate the package on CRAN and then download the package source to your local machine. For example:

```
$ wget http://cran.r-project.org/src/contrib/<package_name_version>.tar.gz
```

Then, install the package using the command **R CMD INSTALL**:

```
$ R CMD INSTALL <package_name_version>.tar.gz
```

NOTE: A major difference between installing R packages using the R package installer at the R command line and shell command line is that package dependencies must be resolved manually at the shell command line. Package dependencies are listed in the *Depends* section of the package's CRAN site. If dependencies are not identified and installed prior to the package's installation, **you will get an error.**

Furthermore, since you don't run R as root (you don't have permission to write packages into the default system-wide location), when installing, you will be prompted to create a personal library accessible by your userID. You can accept the personal library path chosen by R *(see 'gplots' example below)*, **or specify the library location by passing parameters to the install.packages function as shown below**:

```
> install.packages('<package_name>', lib="/path/to/packages")
```

Once you have installed the library of your choice, you may load it with the library() command as follows:

```
> library('<package_name>')
# Alternatively, you may specify the path if required:
> library('<package_name>', lib.loc="/path/to/packages")
```

INSTALLING PACKAGES – *STEP-BY-STEP EXAMPLE*

The following example assumes you have sourced the proper R version (the example will use version 3.3.1).

a) Start R by using the 'R' Command

```
$ R
```

The R program begins, with a banner.

b) Request to install a package, i.e. 'gplots'

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

```
> install.packages("gplots")
Warning in install.packages("gplots") :
  'lib = "/auto/usc/R/3.3.1/lib64/R/library"' is not writable
Would you like to use a personal library instead?  (y/n)
```

Figure 15: Warning Message after Install Request

As mentioned before, you don't have permission to write packages into the default system-wide location, which is the cause of this Warning message. **Say 'y' twice** to accept using a default personal library. You will then be prompted to select a CRAN mirror. In order for a list to display as shown in Figure 16, you must have X11 Forwarding enabled (*please see section 2 – Connecting to the HPC Network for details*). Select '**USA (CA 1)'** and proceed.

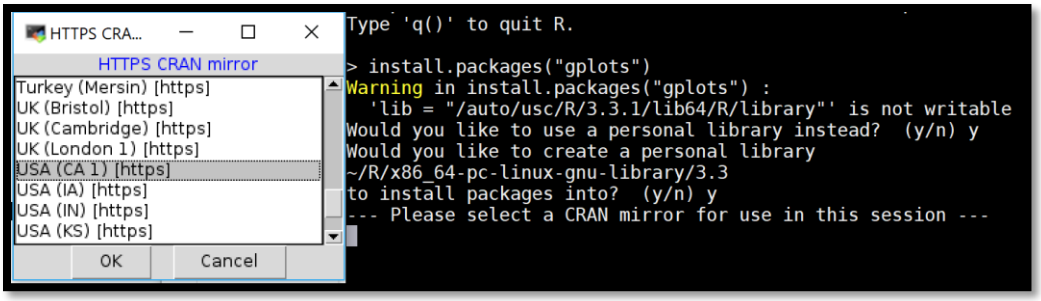

Figure 16: Warning Message after Install Request

At this point, allow for the package installation to proceed. Once complete, you should see the following lines at the end:

```
* DONE (gplots)

The downloaded source packages are in
        '/tmp/Rtmp4ZYgJA/downloaded_packages'
```

Figure 17: Final Lines of Output after Package Installation (for Reference)

c) Check the Installed Packages in your personal library

To check the package has been installed successfully, you need to locate it at the personal library that was created by default when prompted. *See the output of the search in figure 18.*

```
> installed.packages(lib.loc="~/R/x86_64-pc-linux-gnu-library/3.3")
```

```
> installed.packages(lib.loc="~/R/x86_64-pc-linux-gnu-library/3.3")
        Package   LibPath                                 Version  Priority
bitops  "bitops"  "~/R/x86_64-pc-linux-gnu-library/3.3" "1.0-6"  NA
caTools "caTools" "~/R/x86_64-pc-linux-gnu-library/3.3" "1.17.1" NA
gdata   "gdata"   "~/R/x86_64-pc-linux-gnu-library/3.3" "2.18.0" NA
gplots  "gplots"  "~/R/x86_64-pc-linux-gnu-library/3.3" "3.0.1"  NA
gtools  "gtools"  "~/R/x86_64-pc-linux-gnu-library/3.3" "3.5.0"  NA
        Depends        Imports                                    LinkingTo
bitops  NA             NA                                         NA
caTools "R (>= 2.2.0)" "bitops"                                   NA
gdata   "R (>= 2.3.0)" "gtools, stats, methods, utils"            NA
gplots  "R (>= 3.0)"   "gtools, gdata, stats, caTools, KernSmooth" NA
gtools  "R (>= 2.10)"  NA                                         NA
        Suggests      Enhances License      License_is_FOSS
bitops  NA            NA       "GPL (>= 2)" NA
caTools "MASS, rpart" NA       "GPL-3"      NA
gdata   "RUnit"       NA       "GPL-2"      NA
gplots  "grid, MASS"  NA       "GPL-2"      NA
gtools  NA            NA       "GPL-2"      NA
        License_restricts_use OS_type MD5sum NeedsCompilation Built
bitops  NA                    NA      NA     "yes"            "3.3.1"
caTools NA                    NA      NA     "yes"            "3.3.1"
gdata   NA                    NA      NA     "no"             "3.3.1"
gplots  NA                    NA      NA     "no"             "3.3.1"
gtools  NA                    NA      NA     "yes"            "3.3.1"
```

Figure 18: Installed Packages Snapshot

You may notice R has also installed all required dependencies for gplots: 'bitops', 'gtools', 'gdata', and 'caTools'.

A faster way to check whether a specific library has been installed is to call system.file() with the package name:

```
> system.file(package="<package_name>")
```

```
> system.file(package="gplots")
[1] "/auto/rcf-proj/hpcc/hachuelb/myRPackages/x86_64-pc-linux-gnu-library/3.3/gplots"
```

Figure 19: system.file() example

c) Load the Library

Once the package has been properly installed, you may load the library.

```
> library("gplots")
Alternatively, you may specify the path if required:
> library("gplots", lib.loc="~/R/x86_64-pc-linux-gnu-library/3.3")
```

```
> library("gplots", lib.loc="~/R/x86_64-pc-linux-gnu-library/3.3")

Attaching package: 'gplots'

The following object is masked from 'package:stats':

    lowess
```

Figure 20: Loading a Library (Output Message)

c) Use the Library

Once the package has been properly loaded, you may use it. *We will follow a quick example from https://cran.r-project.org/web/packages/gplots/gplots.pdf/*

Type the following into the command line:

INPUT:

```
# create a vector with some values and long labels
values <- sample(1:10)
names(values) <- sapply(letters[1:10], function(x) paste(rep(x, 10), sep="",collapse=""))
# barplot labels are too long for the available space, hence some are not plotted
barplot(values)
```

```
> values <- sample(1:10)
> names(values) <- sapply(letters[1:10], function(x) paste(rep(x,10), sep="",collapse=""))
> barplot(values)
```

Figure 21: Sample Input to use gplots Package

OUTPUT:

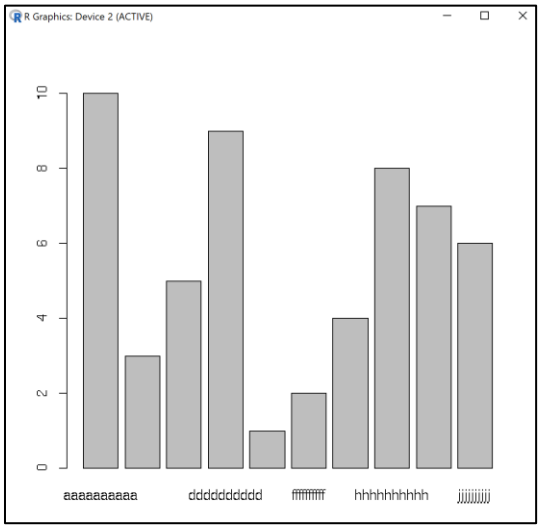

Figure 22: gplots barplot function sample output (with X11 forwarding)

NOTE: Next time you log into the HPC network, please ensure you source the same version of R for which you installed the packages (if you wish to use them). Furthermore, when you start R, you will need to import the library you installed using the **library()** command, as shown above in step c.

If you installed the package from the command line (i.e., not from within R), R may have to specify an absolute path to the library (the second line of step c).

SEARCH PATHS FOR PACKAGES

R packages are installed into *libraries*, which are directories in the file system containing a subdirectory for each package installed there.

In order to display you library paths (all the library 'trees' R knows about), you may run the following command during your R session:

```
> .libPaths()
```

```
> .libPaths()
[1] "/auto/rcf-proj/hpcc/hachuelb/myRPackages/x86_64-pc-linux-gnu-library/3.3"
[2] "/auto/usc/R/3.3.1/lib64/R/library"
```

Figure 23: .libPaths() sample output within R session

Users can have one or more libraries, normally specified by the environment variable `R_LIBS_USER`. This has a default value (to see it, use '`Sys.getenv("R_LIBS_USER")`' within an R session), but that is only used if the corresponding directory actually exists (which by default it will not).

You may set R's library path with the `R_LIBS_USER` environment variable or in **.Rprofile**:

a) Add the following line to your **env.sh** or **PBS script**:

```
$ export R_LIBS_USER=~/R/x86_64-pc-linux-gnu-library/3.3:${R_LIBS_USER}
```

b) Create an **.Rprofile** file in your home directory and add path:

```
# create a new file, name it .Rprofile
$ touch .Rprofile
# open an editor, such as nano, to edit the newly created .Rprofile file
$ nano .Rprofile
# Add the following line to the newly created .Rprofile file, save and close
libPaths(c(.libPaths(), "~/R/x86_64-pc-linux-gnu-library/3.3"))
```

REMOVING PACKAGES

Packages can be removed in a number of ways. From the command line they can be removed through the following command:

```
$ R CMD REMOVE -l /path/to/library <package_name_1> <package_name_2> ...
```

From an active R session they can be removed through the following command:

```
> remove.packages(c("<package_name_1> ", "<package_name_2> "), lib =
file.path("path", "to", "library"))
```

You may also simply remove the package directory from the library.

V. PARALLEL PROGRAMMING IN R

REMINDER: To run your jobs, you must use the scheduler to allocate yourself a compute node, rather than running R on the login node. For guidance on how to do so, please refer to the ***HPC Advance Topics*** course, or contact your HPC POC.

It is important to note that not all algorithms can or should be parallelized. See below for a quick reference:

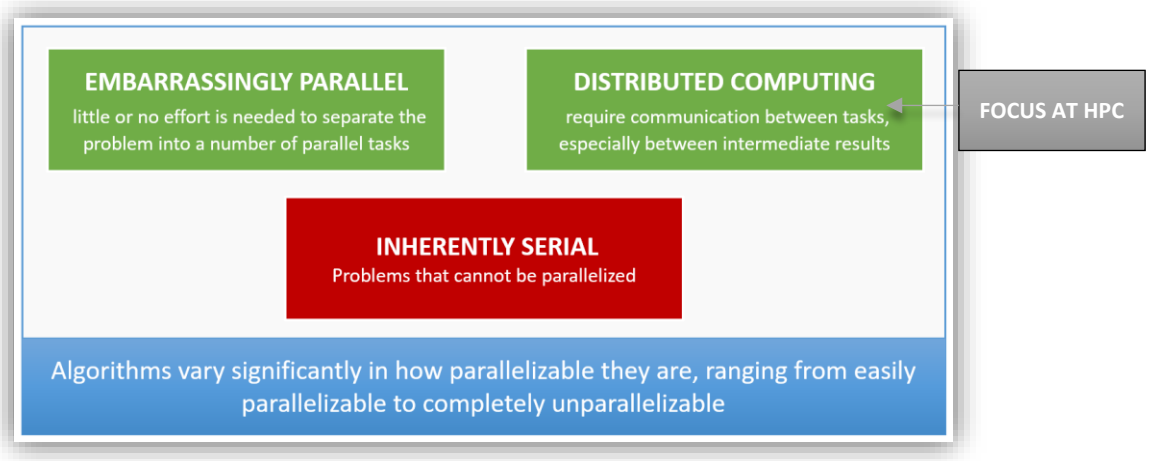

Figure 24: Types of Problems/Algorithms in Parallel Programming

R PARALLEL PROGRAMMING PACKAGES

The following libraries that support parallel processing are installed in the most recent version of R under HPC: parallel, snow and snowfall. For the purposes of this guide, we will be using the **parallel** package.

The parallel package was first included in R 2.14.0. It builds on the work done for CRAN packages multicore and snow and provides drop-in replacements for most of the functionality of those packages.

KEY CONCEPT: `lapply()` Function

A key tool to understand how we may parallelize in R is the **lapply(x)** function.

The function only takes **a single parameter** (a list x), and returns a list of the same size, each element of which is the result of applying the **given function** to the corresponding element of the list given as parameter x:

Run an instance of R and type the following:

```
> lapply(1:5, function(x) c(x, x^2, x^3))
```

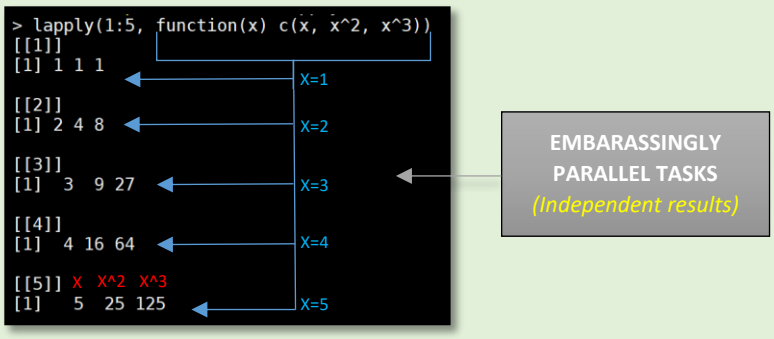

Figure 25: Sample lapply() output

You can use R's **'parallel' packages** to perform the above but in an actual parallel manner.

LAPPLY-BASED PARALLELISM

Lapply-based parallelism is the most intuitive way to parallelize your algorithms in R, as it is an extension of the above lapply() function.

The parallel package will allow us to perform operations such as the one described above where the main modification is that we need to start with setting up a cluster, a collection of "workers" that will be doing the job.

As analogous to **lapply()** there are:

```
> mclapply(X, FUN, ..., mc.cores)
> parLapply(cl, x, FUN, ...)
```

MCLAPPLY: SHARED-MEMORY PARALLELISM

The "mc" stands for "multicore". This function distributes the lapply tasks across multiple CPU cores to be executed in parallel.

Using shared memory (multicore) tends to be much simpler because all parallel tasks automatically share their R environments and can see the objects and variables we defined before the parallel region that is encapsulated by mclapply(). [5]

As with almost every form of parallel programming, you've got to take one step backwards to make two steps forwards. In a practical sense, that means making your program a little more complicated so that parallelizing it becomes possible. **Let's take our above function as an example.** In order to compare the two, we will *'re-design'* the above lapply() function as follows:

```
#!/usr/bin/env Rscript

#sequential version of function shown in prior note
sequential.function <- function(i) {
    #the print function will print:
    #the unique parallel worker id and the hostname on which that worker is running
    sprintf('LOOP ITERATION: %d RUNNING ON WORKER_ID: %d AND NODE: %s WHERE: x= %d, x^2= %d and x^3= %d',
       i, Sys.getpid(), Sys.info()[c("nodename")], i, i^2, i^3);
}

#run the lapply function (sequential)
lapply(1:10, FUN=sequential.function)
```

Figure 26: lapply() function re-write for comparison with mclapply()

Running the above code on HPC will give us the following output:

```
hachuelb@hpc-login3:~/R$ Rscript r_lapply.R
[[1]]
[1] "LOOP ITERATION: 1 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 1, x^2= 1 and x^3= 1"

[[2]]
[1] "LOOP ITERATION: 2 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 2, x^2= 4 and x^3= 8"

[[3]]
[1] "LOOP ITERATION: 3 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 3, x^2= 9 and x^3= 27"

[[4]]
[1] "LOOP ITERATION: 4 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 4, x^2= 16 and x^3= 64"

[[5]]
[1] "LOOP ITERATION: 5 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 5, x^2= 25 and x^3= 125"

[[6]]
[1] "LOOP ITERATION: 6 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 6, x^2= 36 and x^3= 216"

[[7]]
[1] "LOOP ITERATION: 7 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 7, x^2= 49 and x^3= 343"

[[8]]
[1] "LOOP ITERATION: 8 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 8, x^2= 64 and x^3= 512"

[[9]]
[1] "LOOP ITERATION: 9 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 9, x^2= 81 and x^3= 729"

[[10]]
[1] "LOOP ITERATION: 10 RUNNING ON WORKER_ID: 8658 AND NODE: hpc-login3 WHERE: x= 10, x^2= 100 and x^3= 1000"
```

Figure 27: Output from lapply() code in figure 26 on the HPC network

As you can see from the above output, every loop iteration has run on the same core (*worker_id: 8658*, in this example). We'd like to make use of the mclapply() function to use our available cores in parallel. In order to do so, we will update our code to load the 'parallel' library, detect the amount of available cores, and pass this number to the mclapply() function for parallel processing:

```
#!/usr/bin/env Rscript

#parallelizable version of function shown in prior note
parallel.function <- function(i) {
    #the print function will print the unique parallel worker id and the hostname on which that worker is running
    sprintf('LOOP ITERATION: %d RUNNING ON WORKER_ID: %d AND NODE: %s WHERE: x= %d, x^2= %d and x^3= %d',
       i, Sys.getpid(), Sys.info()[c("nodename")], i, i^2, i^3);
}
#Load the 'parallel' library
library(parallel)
#detect the number of available cores
cores <- detectCores()
#run the mclapply function and assign all available cores detected
mclapply(1:10, FUN=parallel.function, mc.cores = cores)
```

Figure 28: mclapply() code for parallel processing on all available cores

Running the above code on HPC will give us the following output:

```
hachuelb@hpc-login3:~/R$ Rscript r_mclapply.R
[[1]]
[1] "LOOP ITERATION: 1 RUNNING ON WORKER_ID: 13634 AND NODE: hpc-login3 WHERE: x= 1, x^2= 1 and x^3= 1"

[[2]]
[1] "LOOP ITERATION: 2 RUNNING ON WORKER_ID: 13635 AND NODE: hpc-login3 WHERE: x= 2, x^2= 4 and x^3= 8"

[[3]]
[1] "LOOP ITERATION: 3 RUNNING ON WORKER_ID: 13636 AND NODE: hpc-login3 WHERE: x= 3, x^2= 9 and x^3= 27"

[[4]]
[1] "LOOP ITERATION: 4 RUNNING ON WORKER_ID: 13637 AND NODE: hpc-login3 WHERE: x= 4, x^2= 16 and x^3= 64"

[[5]]
[1] "LOOP ITERATION: 5 RUNNING ON WORKER_ID: 13638 AND NODE: hpc-login3 WHERE: x= 5, x^2= 25 and x^3= 125"

[[6]]
[1] "LOOP ITERATION: 6 RUNNING ON WORKER_ID: 13639 AND NODE: hpc-login3 WHERE: x= 6, x^2= 36 and x^3= 216"

[[7]]
[1] "LOOP ITERATION: 7 RUNNING ON WORKER_ID: 13640 AND NODE: hpc-login3 WHERE: x= 7, x^2= 49 and x^3= 343"

[[8]]
[1] "LOOP ITERATION: 8 RUNNING ON WORKER_ID: 13641 AND NODE: hpc-login3 WHERE: x= 8, x^2= 64 and x^3= 512"

[[9]]
[1] "LOOP ITERATION: 9 RUNNING ON WORKER_ID: 13642 AND NODE: hpc-login3 WHERE: x= 9, x^2= 81 and x^3= 729"

[[10]]
[1] "LOOP ITERATION: 10 RUNNING ON WORKER_ID: 13643 AND NODE: hpc-login3 WHERE: x= 10, x^2= 100 and x^3= 1000"
```

Figure 29: Output from mclapply() code in figure 28 on the HPC network

The downside is that this shared memory approach to parallelism in R is limited by how many cores your computer has. Modern supercomputers (such as HPC) achieve parallelism via distributed memory, meaning many nodes do not share memory and need to be explicitly given all of the variables and objects that were created outside of the parallel region. [5]

PARLAPPLY: DISTRIBUTED-MEMORY PARALLELISM

To use more than one node's worth of CPU cores with lapply-style parallelism, you have to use some type of networking so that each node can communicate with each other and shuffle the relevant data around. As such, there's a bit more bookkeeping involved with using a distributed memory version of lapply, but

fortunately, the actual logic of your application doesn't need to change much. Some important points are listed below:

1. We must create a "cluster" object using the **makeCluster()** function. This "cluster" will be what determines what nodes and cores the **parLapply()** function will have available for distributing work.

2. Because we are using distributed memory, not all of our worker CPUs can see the data we have loaded into the main R script's memory. Thus, we need to explicitly distribute that data to our worker nodes using the **clusterExport()** function.

We will modify our prior example in order to properly demonstrate the need to use clusterExport():

```
#!/usr/bin/env Rscript

#parallelizable version of function shown in prior note
parallel.function <- function(i) {
    #the print function will print the unique parallel worker id and the hostname on which that worker is running
    sprintf('LOOP ITERATION: %d RUNNING ON WORKER_ID: %d AND NODE: %s WHERE: (base)^i= %d',
       i, Sys.getpid(), Sys.info()[c("nodename")], base^i);
}

#Load the 'parallel' library
library(parallel)
#detect the number of available cores
cores <- detectCores()
#store the base for the exponent (for testing clusterExport)
base <- 2
# Initiate cluster with the number of detected cores
cl <- makeCluster(cores)
#export the cluster to explicitly distribute data (base for exponent) to the worker nodes
clusterExport(cl, "base")
#run the parlapply function and pass the cluster as the first parameter
parLapply(cl,1:10, fun=parallel.function)
#tear down the cluster at the end of our script
stopCluster(cl)
```

Figure 30: parLapply() code for parallel processing

As shown above, note that you need the **clusterExport(cl, "base")** in order for the function to see the *base variable*. If you are using some special packages you will similarly need to load those.

Running the above code on HPC will give us the following output:

NOTE

Because we have run this tutorial on the head node, the output similarly shows (as with mclapply) that we have made use of cores within a single node (head node) only.

```
hachuelb@hpc-login3:~/R$ Rscript r_parlapply.R
[[1]]
[1] "LOOP ITERATION: 1 RUNNING ON WORKER_ID: 44226 AND NODE: hpc-login3 WHERE: (base)^i= 2"

[[2]]
[1] "LOOP ITERATION: 2 RUNNING ON WORKER_ID: 44238 AND NODE: hpc-login3 WHERE: (base)^i= 4"

[[3]]
[1] "LOOP ITERATION: 3 RUNNING ON WORKER_ID: 44259 AND NODE: hpc-login3 WHERE: (base)^i= 8"

[[4]]
[1] "LOOP ITERATION: 4 RUNNING ON WORKER_ID: 44283 AND NODE: hpc-login3 WHERE: (base)^i= 16"

[[5]]
[1] "LOOP ITERATION: 5 RUNNING ON WORKER_ID: 44307 AND NODE: hpc-login3 WHERE: (base)^i= 32"

[[6]]
[1] "LOOP ITERATION: 6 RUNNING ON WORKER_ID: 44319 AND NODE: hpc-login3 WHERE: (base)^i= 64"

[[7]]
[1] "LOOP ITERATION: 7 RUNNING ON WORKER_ID: 44337 AND NODE: hpc-login3 WHERE: (base)^i= 128"

[[8]]
[1] "LOOP ITERATION: 8 RUNNING ON WORKER_ID: 44358 AND NODE: hpc-login3 WHERE: (base)^i= 256"

[[9]]
[1] "LOOP ITERATION: 9 RUNNING ON WORKER_ID: 44379 AND NODE: hpc-login3 WHERE: (base)^i= 512"

[[10]]
[1] "LOOP ITERATION: 10 RUNNING ON WORKER_ID: 44388 AND NODE: hpc-login3 WHERE: (base)^i= 1024"
```

Figure 31: Output from parLapply() code in figure 30 on the HPC network

APPENDIX: CITATIONS

[1] "What Is R?" *R*, The R Foundation, www.r-project.org/about.html.

[2] *R Installation and Administration*, cran.r-project.org/doc/manuals/r-release/R-admin.html#Add_002don-packages.

[3] "R Enterprise Installation and Administration Guide." *R Package Installation Basics*, 31 Aug. 2017, docs.oracle.com/cd/E83411_01/OREAD/R-package-installation-basics.htm#OREAD-GUID-064D5832-664A-4E93-AFCC-1A3B70D001AC.

[4] "How-to Go Parallel in R – Basics + Tips." *G-Forge*, 16 Feb. 2015, gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/.

[5] Lapply-Based Parallelism, www.glennklockwood.com/data-intensive/r/lapply-parallelism.html.