**Parallel processing and the bootstrap**

Embarrassingly parallel

Monte Carlo simulations involve repeating the same calculation R times AND what happens on iteration r does not affect what happens on iteration r′ for r ≠ r′. These types of simulations fall under the situation often referred to as “embarrassingly parallel” because each iteration can be completed at the same time by separate processors.

Multicore processors

The most common form of parallel processing used to involve running a program on multiple computers at the same time. This involved a master computer sending out instructions to “worker” (or “slave”) computers. Once these instructions were completed, the master computer would combine calculations from the workers into one usable form. Below is a diagram of what this may look like for one master and four workers.



Thus, a master computer may be in control of a 1,000 data set simulation where 250 separate data sets are sent to each of the four workers. Note that a master could also be a worker as well.

The advent of multicore processors in personal computers around 2006(?) made parallel processing more accessible. Now, the same type of methods above can be applied to one computer that has multiple cores within its processor.

Intel processors take multicore processors even further by introducing the concept of “threads”. A multithreaded core allows for more than one set of operations to be controlled by a single core. Thus, a two core computer may have two threads per core. More on how this affects computation time shortly.

Example: My Fujitsu tablet PC

This has an Intel i5 processor where its two cores each run at 2.40GHz. Below are some of its specifications.



Additional information from Dell at [http://ark.intel.com/
products/47341/Intel-Core-i5-520M-Processor-3M-Cache-2\_40-GHz](http://ark.intel.com/products/47341/Intel-Core-i5-520M-Processor-3M-Cache-2_40-GHz).



There are two cores each with two threads. Below is the information given by R and its parallel package when I want it to detect my cores:

> library(package = parallel)

> detectCores()

[1] 4

> detectCores(logical = FALSE)

[1] 2

Notice that R by default provides the TOTAL number of threads available.

Example: My Dell desktop computer

This has an AMD Phenom II X6 1090T Processor where each core runs at 3.2GHZ. There are six cores with 1 thread per core.

> library(parallel)

> detectCores()

[1] 6

Example: Tusker at the Holland Computing Center (HCC)

This is a UNL supercomputer available to faculty and students for research. From [http://hcc.unl.edu/facilities/
index.php#tusker](http://hcc.unl.edu/facilities/index.php#tusker):





A node is a section of a supercomputer that runs one operating system. Each node has 64 cores. This means there are (106 nodes × 64 cores) = 6,784 total cores.

Below is what happens when using *detectCores()* in R:

> library(package = parallel)

> detectCores()

[1] 64

One can access other cores beyond those in one node.

Computation time

In the diagram on p. 9.2, one would think that the use of four workers allows the computation time to be ¼ of the time it would take on a single computer. This is not exactly the case because the master still needs to combine all of the calculations at the end. Also, the amount of communication needed between the master and workers can take time as well. Thus, the amount of computation time should be at least a little greater than ¼.

When cores have multiple threads, figuring the computation time savings is not as easy. There is still only one calculation that can be performed at one time on one core. Where a time savings may come in is when there is waiting time between calculations. Rather than the core not being used during the wait time, another thread can use the core. Generally, I have found the following:

* A k-core processor with 2 threads per core is faster than a k-core processor with only 1 thread per core.
* A k-core processor with 2 threads per core is MUCH slower than a 2k-core processor with 1 thread per core.

Parallel package

An overview of parallel computing in R can be found at the CRAN task view for it at [http://cran.r-project.org
/web/views/HighPerformanceComputing.html](http://cran.r-project.org/web/views/HighPerformanceComputing.html)

The parallel package made its debut in 2011 with R 2.14.0, and it is automatically installed in R (still need to run *library(parallel)* first). The package takes code from two other packages (snow and multicore) that had been in development for a few years prior to 2011.

A vignette on parallel is installed with R. On my computer, this is at [http://127.0.0.1:27186/
library/parallel/doc](http://127.0.0.1:27186/library/parallel/doc) (you can go to R’s HTML help to find it too). Examples of how to use the package with respect to the bootstrap are given toward the end of the manual! These examples are what provided the motivation for my own examples given next.

 Example: Air conditioning data (pp\_AC.R)

While this example does not need to use parallel processing, it serves as a simple example to illustrate the code and allow us to perform some checks.

Without parallel processing (Fujitsu Tablet PC):

> options(width = 60)

> y<-c(3,5,7,18,43,85,91,98,100,130,230,487)

> t<-mean(y)

> cat("My sample is", sort(y), "\n which produces an

 observed statistic of", t, "\n")

My sample is 3 5 7 18 43 85 91 98 100 130 230 487

 which produces an observed statistic of 108.0833

> library(boot)

> calc.t<-function(data, i) {

 d<-data[i]

 mean(d)

 }

> #Try it

> calc.t(data = y, i = 1:length(y))

[1] 108.0833

> R.total<-1000000

> #Find start time

> start.time<-proc.time()

> set.seed(9182)

> boot.res1<-boot(data = y, statistic = calc.t, R =

 R.total, sim="ordinary")

> boot.res1

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:

boot(data = y, statistic = calc.t, R = R.total, sim = "ordinary")

Bootstrap Statistics :

 original bias std. error

t1\* 108.0833 -0.0327915 37.64411

> #Find end time and total time elapsed

> end.time<-proc.time()

> save.time<-end.time-start.time

> cat("\n Number of minutes running:", save.time[3]/60, "\n

 \n")

 Number of minutes running: 0.3953333

> nrow(boot.res1$t)

[1] 1000000

Notice that I use 1,000,000 resamples! This large number of resamples would not be necessary in application! It was only done to make sure the code would take some time to complete.

With parallel processing (Fujitsu Tablet PC):

> library(parallel)

> #Opens additional R sessions in background to enable

 running in parallel

> number.cores<-2

> cl<-makeCluster(spec = number.cores) #Cluster of size 2

> #Note: need to put library(boot) and calc.t() function

 inside of here because it is sent to new R sessions

> per.core<-function(y) {

 y<-c(3,5,7,18,43,85,91,98,100,130,230,487)

 R<-500000

 library(boot)

 calc.t<-function(data, i) {

 d<-data[i]

 mean(d)

 }

 boot.per<-boot(data = y, statistic = calc.t, R = R,

 sim="ordinary")

 }

> #Multiple streams of seeds – one set per core being used

> #Notice how 9182 will not produce the same resamples as

 with single core set.seed()

> clusterSetRNGStream(cl = cl, iseed = 9182)

> #Find start time

> start.time<-proc.time()

> boot.res2<-do.call(what = c, args = parLapply(cl = cl, X

 = 1:number.cores, fun = per.core))

> boot.res2

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:

boot(data = y, statistic = calc.t, R = R, sim = "ordinary")

Bootstrap Statistics :

 original bias std. error

t1\* 108.0833 -0.01046458 37.70325

> #Find end time and total time elapsed

> end.time<-proc.time()

> save.time<-end.time-start.time

> cat("\n Number of minutes running:", save.time[3]/60, "\n

 \n")

 Number of minutes running: 0.2148333

> stopCluster(cl) #Closes additional R sessions

> nrow(boot.res2$t)

[1] 1000000

Comments about the code:

* When I first created this code, I tried to pass in values of *y* and *R* into the *per.core()* function because *parLapply()* has “…” in its syntax. However, I received an error message when I tried it. Thus, it appears one needs to actually put in the data (or read it in from an external file) directly in the *per.core()* function. Similarly, the same needs to be done with the number of resamples PER core.
* The *parLapply()* function works like the *apply()* and *lapply()* function. The “L” instructs R to return the results from each call to *per.core()* in a list format. This is convenient because the *boot()* function at the end of *per.core()* returns a list. I recommend running the *parLapply()* function outside of the do.call() function to see what happens (two lists are returned where each contains the results from a core). Below are additional comments about the arguments:
	+ The *cl* argument specifies the number of cores for parallel processing. Thus, the *per.core()* function is implemented this many times.
	+ The *X* argument of *lapply()* gives the data to be summarized. From its help,

> x <- list(a = 1:10, beta = exp(-3:3), logic =

 c(TRUE,FALSE,FALSE,TRUE))

> # compute the list mean for each list element

> lapply(x,mean)

$a

[1] 5.5

$beta

[1] 4.535125

$logic

[1] 0.5

For our implementation of *parLapply()*, the *X* argument really does not play a role. However, it needs to have a length of *number.cores*. For example, if *X = 1:3*, R runs *per.core()* twice on one core and once on another core.

* The *do.call()* function instructs R to use the *c()* function when combining the results obtained from each core. Note that the *c()* function is generic so *c.boot()* is actually used. The end result is our usual type of saved results from running *boot()* on a single core. Please note that a *c()* function may not work like this for non-*boot()* function implementations.

The resamples with and without parallel processing are not the same:

> save.ind1<-boot.array(boot.out = boot.res1, indices =

 TRUE)

> save.ind1[1,]

 [1] 9 8 4 2 3 3 1 7 9 11 11 7

> save.ind2<-boot.array(boot.out=boot.res2, indices=TRUE)

> save.ind2[1,]

 [1] 1 9 8 7 7 5 3 1 8 6 3 5

How do you know that your computer is actually using multiple cores?

Watch a Windows gadget that monitors your processor. For example, below is a screen capture taken while running the previous code.



Approximately 50% of my processing capability is being used. This gadget counts each thread separately, so this is why 50% rather than 100% is given.

How do you know that *makeCluster()* actually opens other R sessions?

Start the Task Manager and look at the number of R processes running. Below is a screen capture from running the previous code.



Examination of time when asking R to use 1, 2, …, 6 cores.



Obviously, there should be no gains from using 5 or 6 cores with my computer. We see some time savings from specifying 3 or 4 cores in comparison to the 2 cores that the computer actually has. Thus, there is a small benefit from Intel’s multi-thread capabilities. Of course, note that when I specified 4 or more cores, I was unable to use my computer for other tasks very well because all of its processing power was already being used.

In general, there may be times when you want to stop code from running. What happens if you select the “stop current calculation” button or the ESC key when using the parallel package?

The other R sessions will not stop! In fact, if you use *stopcluster()* right afterward, the R sessions are still not closed:

> stopCluster(cl)

Error in summary.connection(connection) : invalid connection

To terminate the R sessions, you will need to either

* + Let them finish
	+ Terminate the R sessions through using the Windows Task Manager

foreach package

The foreach package allows for parallel processing as well, and it was developed by Revolution Analytics. The package is not automatically installed in R, so you will need to install it yourself. A vignette on the package is available at [http://cran.r-project.org/web/packages/
foreach/index.html](http://cran.r-project.org/web/packages/foreach/index.html).

The *foreach()* function in the package operates like the *for()* function, but it distributes parts of the computations to different cores. The syntax of the function is

*foreach(i = 1:R , .combine = \_\_\_ ) %dopar {*

*<code for each core>*

*}*

Comments:

* The *.combine* argument specifies how the results from the different cores should be combined. For example, the *c* function can be specified to simply put the results together into a vector. The *cbind* function puts results together into a matrix. The *list* function puts the results into a list.
* Objects can be created outside of *foreach()* and called from within the function. For example, data can be read into R outside of the function call and simply called its object name inside the function call.
* A *.package* argument can be used to have packages loaded within each worker. The package names need to be within quotes.

Along with the foreach package, Revolution Analytics has written a package named doParallel. This package allows one to make the multiple cores ready for use.

Example: Air conditioning data (pp\_AC.R)

Below is a for loop implementation of resampling with R = 1000:

> y<-c(3,5,7,18,43,85,91,98,100,130,230,487)

> R<-1000

> start.time<-proc.time()

> #Regular for loop

> set.seed(9182)

> save.res1<-numeric(length = R)

> for(i in 1:R) {

 ind<-sample(x = 1:length(y), size = length(y), replace

 = TRUE)

 save.res1[i]<-mean(x = y[ind])

 }

> #A potential statistic of interest involving the t\*'s

> var(save.res1)

[1] 1327.614

> end.time<-proc.time()

> save.time<-end.time-start.time

> cat("\n Number of minutes running:", save.time[3]/60, "\n

 \n")

 Number of minutes running: 0.002833333

Next is a *foreach()* function implementation using two cores:

> library(doParallel)

> cl<-makeCluster(spec = 2) #Don't need to load parallel

 package because doParallel does it

> registerDoParallel(cl = cl)

> start.time<-proc.time()

> clusterSetRNGStream(cl = cl, iseed = 9182) #Multiple

 streams of seeds

> save.res2<-foreach(i = 1:R, .combine = c) %dopar% {

 ind<-sample(x = 1:length(y), size = length(y), replace =

 TRUE)

 mean(x = y[ind])

 }

> #A potential statistic of interest involving the t\*'s

> var(save.res2)

[1] 1440.427

> stopCluster(cl)

> end.time<-proc.time()

> save.time<-end.time-start.time

> cat("\n Number of minutes running:", save.time[3]/60, "\n

 \n")

 Number of minutes running: 0.2508333

The *foreach()* function is much slower than the *for()* function! I was quite surprised at first! As a verification of the results, I ran *foreach()* again with one core, and the code took 0.42 minutes. I looked into the problem a little more, and it appears others have noticed this as well. There is supposedly a lot of time spent with the combination of the results at the end. Some users recommend *foreach()* only when there are a few combinations of items.

Motivated by how the parallel package code example was set-up, I reconstructed my code as follows:

> cl<-makeCluster(spec = 2)

> registerDoParallel(cl = cl)

> calc.t<-function(data, i) {

 d<-data[i]

 mean(d)

 }

> R<-1000

> start.time<-proc.time()

> clusterSetRNGStream(cl = cl, iseed = 9182)

> save.res3<-foreach(i = 1:2, .combine = c, .packages =

 "boot") %dopar% {

 boot.per<-boot(data = y, statistic = calc.t, R = R/2,

 sim="ordinary")

 boot.per$t

 }

> #A potential statistic of interest involving the t\*'s

> var(save.res3)

[1] 1342.821

> stopCluster(cl)

> end.time<-proc.time()

> save.time<-end.time-start.time

> cat("\n Number of minutes running:", save.time[3]/60, "\n

 \n")

 Number of minutes running: 0.006

This is much better, but not quite as good as the *for()* function. To examine this further, I used larger R’s as shown in the table below

|  |  |  |
| --- | --- | --- |
| **R** | ***for()*** | ***foreach()*, 2 cores** |
| 1,000 | 0.0028 | 0.0060 |
| 10,000 | 0.0457 | 0.0242 |
| 1,000,000 | 0.4527 | 0.2278 |

The R = 1,000,000 time for *foreach()* is similar to what was obtained earlier with the parallel package.

Note that *.combine = list* could be used as well in *foreach()*. However, I have not been able to get a *do.call()* like implementation as in the parallel package code to combine the results nicely. Of course, one could write their own small function to do it ☺.

HCC computers

The Holland Computing Center (HCC) has a few supercomputers available for student and faculty use. The next discussion focuses on their newest computer, named Tusker.

Tusker uses a LINUX operation system that is just text based (no windows, no mouse). All commands must be entered via a command prompt. Below are some notes on its use:

* Request an account at <http://hcc.unl.edu/newusers>. Students will need to be part of a research group (can be requested by a faculty member).
* Putty is a decent terminal interface program for Tusker. You can download it from <http://www.putty.org/>. Simply run the executable file (putty.exe) whenever you want to use it (the program does not need to be “installed”).
* WinSCP is a FTP interface program for Tusker (use it to transfer files, like programs, to and from Tusker).
* All programs should be run in the work folder of your research group. Note that this is not backed up.

Next is an example session to run some of the same code we used earlier in the parallel package example.

1. Log into Tusker via Putty and get into work directory.

Open Putty:



Notice the address is tusker.unl.edu. Also, I have saved this information in my “tusker” session setting.

I log in and change directories to get to my work directory:



Notice the “ls” command allows you to see the files in the directory – kind of like R’s use of *ls()* ☺.

1. Transfer files via WinSCP and get into work directory

I opened the program and used tusker.unl.edu as the host name with the same login and password as in Putty. Below is my window when I first log on to Tusker.



Again, I need to change directories to get into my work directory. This is done as you would normally in Windows (go down three folders until you find “work” and then go up to the folder with the same name as your user name).



Files can now be dragged and dropped into Tusker.

1. If needed, you can edit files on Tusker via the program “nano” by typing “nano <FILE NAME>” at a command prompt without the quotes and without <>.

I opened my file detectcores.txt script file:



1. R is run in batch mode through using a script file. The above file in nano shows one batch file. Below are comments about the commands:
	1. PBS corresponds to the “public scheduling system”. All programs are submitted to the system and run as deemed possible by PBS. Typing “man PBS” without the quotes at a command prompt will give you more information about it. Type “q” to quit the manual.
	2. “pp” is the “N”ame that I have designated for the process to be run. Note that this is just an identifier, and it does not need to be the actual R program name (this is given later).
	3. “select” allows you to select the number of cores to use. Because I was doing some benchmarking, I had mine set high in this file. I recommend setting the number of cores to the actual number of cores you want to use.
	4. “walltime” is the maximum number of hours:minutes:seconds you are giving your program to finish. If it does not finish within the allotted time, PBS will terminate it.
	5. “pp.stdout” and “pp.stderr” are files that correspond to your script for output and errors, respectively. Unless you are using a script file much different from mine, nothing will go into these files.
	6. “R CMD BATCH pp.r” tells PBS to run the pp.r program in batch mode. All output is sent to the file pp.r.Rout.
2. The script file is run using the following:



The “qsub” command submits a program to the queue. PBS determines when to run it. The “qstat” command allows you to see the status of the script. The second jobname “pp” is the script which was submitted.

While not necessary, I used “-m e M bilder@unl.edu” with “qsub” to have Tusker send me an e-“m”ail at the “M”ail address bilder@unl.edu once the program has “e”nded.

Once the script file has been completed, I can use WinSCP to upload pp.r.Rout to my computer.

Comments:

* My pp.r program is available from the course website. It is essentially the same as pp\_AC.R, but set-up to run for 1, 2, 4, 8, 10, 20, and 25 cores. The detectcores.txt script file is available on the R web page of the course website.
* The parallel package can be used in a similar manner as with the previous Windows applications. However, there are other ways available to use the package for parallel processing. For example, see the *mclapply()* function (page 3 of the parallel package vignette).
* I ran pp.r four times where the last time I included code for 20 and 25 cores. Below is a plot of the time it took each to complete by core.



Overall, we see some variability across the 4 runs and the time rarely gets to be less than 0.08 minutes. Possible causes include: 1) PBS – you are the mercy of the scheduling system ☺ and 2) Communication time between the workers and the master.

*boot()* function’s parallel processing arguments

Below is a screen capture from R’s help for *boot()*.







For non-Windows based computers, the *boot()* function can now automatically use parallel processing. The function uses the multicore or snow packages, from which parallel was derived. I would expect that *boot()* will be updated in the near future for the parallel package, so I decided not to try parallel processing with the arguments within *boot()*.

Grid computing

This involves a collection of autonomous computers that are not connected by a central bus.

UNL participates in the The Open Science Grid (<https://www.opensciencegrid.org>). This grid involves a collection of supercomputers from around the world that can be used for parallel processing purposes. Therefore, if the computers at UNL are not enough, there are even more resources available!

Final comments

* If you want to use your computer simultaneously while doing parallel processing, do not use all of the cores for the parallel processing. Otherwise, your computer will be very slow while you use it! I usually leave one core (or one thread) open.
* The parallel package has other functions that may be useful such as *parApply()* and *parRapply()* which work like the *apply()* function.
* If you need to install packages on Tusker, you can not use the same ways as we have in class. Below is an example of how I have done it on Tusker:

> .libPaths(new = "/work/bilder/bilder")

> .libPaths()

[1] "/lustre/work/bilder/bilder"

[2] "/util/opt/R/2.15.0/gcc/4.4.5/64/lib64/R/library"

> install.packages(pkgs = "bibtex", repos =

 "http://streaming.stat.iastate.edu/CRAN/",

 destdir = "/work/bilder/bilder") #Try to install a

 package

Installing package(s) into '/lustre/work/bilder/bilder'

(as 'lib' is unspecified)

trying URL 'http://streaming.stat.iastate.edu/CRAN/src/contrib/bibtex\_0.3-0.tar.gz'

Content type 'application/x-gzip' length 80248 bytes (78 Kb)

opened URL

==================================================

downloaded 78 Kb

\* installing \*source\* package 'bibtex' ...

\*\* package 'bibtex' successfully unpacked and MD5 sums checked

\*\* libs

<OUTPUT EDITED>

installing to /lustre/work/bilder/bilder/bibtex/libs

\*\* R

\*\* inst

\*\* preparing package for lazy loading

\*\* help

\*\*\* installing help indices

\*\* building package indices

\*\* testing if installed package can be loaded

\* DONE (bibtex)

> library(bibtex)

> bib <- read.bib(package = "bibtex")

> bib

R Development Core Team (2009). \_R: A Language and Environment for Statistical Computing\_. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, <URL: http://www.R-project.org>.