**Parallel processing**

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 will be discussed shortly.

Example: My Surface Pro 2

This computer has what is often referred to as a “4th generation processor” or the “Haswell” architecture for its processor. Specifically, the computer has an Intel i5-4200U processor where its two cores each run at 1.6GHz. These cores can go up to 2.3GHZ when needed. Each core has two threads.

Below are some of its specifications. This is found in Windows 8.1 by going to the Control Panel and selecting System.



Under Advanced System Settings > Hardware > Device Manager, I can see the four threads.



In the Task manager, I have the following information as well:



Additional information from Intel at <http://ark.intel.com/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz>.



Below is the information given by R and its parallel package when I ask 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 Fujitsu tablet PC

This computer has what is often referred to as a “3rd generation processor” or the “Ivy Bridge” architecture for its processor. Specifically, the computer has an Intel i5 processor where its two cores each run at 2.40GHz.

Below are some of its specifications. This is found in Windows 7 by selecting Control Panel > Performance Information and Tools > View and Print Detailed Performance



Additional information from Intel 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).



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: Crane and Tusker at the Holland Computing Center (HCC)

This is a UNL supercomputer available to faculty and students for research. From [https://hcc-docs.unl.edu/display/HCCDOC/HCC+Documentation](https://hcc-docs.unl.edu/display/HCCDOC/HCC%2BDocumentation):

| **Cluster** | **Overview** | **Processors** | **RAM** | **Connection** | **Storage** |
| --- | --- | --- | --- | --- | --- |
| **Crane** | 452 node Production-mode LINUX cluster | Intel Xeon E5-2670 2.60GHz 2 CPU/16 cores per node | \*64GB RAM per node | QDR Infiniband | ~1452 TB shared Lustre storage~1.8 TB local scratch per node |
| **Tusker** | 106 node Production-mode LINUX cluster | Opteron 6272 2.1GHz, 4 CPU/64 cores per node | \*\*256 GB RAM per node\*\*\*2 Nodes with 512GB per node  | QDR Infiniband | ~500 TB shared Lustre storage ~500GB local scratch |

A node is a section of a supercomputer that runs one operating system. Essentially, you can kind of think of it as a single computer within the larger super computer. For Tusker, there are 4 Opteron 6272 CPUs per node and each has 16 cores. Thus, there are 64 cores for each node. Because there are 106 nodes, this means there are a total of 416106 = 6784 cores.

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

> library(package = parallel)

> detectCores()

[1] 64

One can access other cores beyond those in one node.

Computation time

In the diagram on p. 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 prior to 2011.

A vignette on parallel is installed with the help for parallel in R.

Example: Estimate true confidence level for a confidence interval (MC\_sim\_PP.R, Run\_on\_each\_core.R)

This example examines again the Monte Carlo simulation study that estimates the true confidence level (coverage) involving confidence intervals for a variance σ2 where Yi ~ N(μ, σ2) for i = 1, …, 9, μ = 2.713333, and σ2 = 4.820575. There are R = 500 data sets simulated under these settings.

From the bootstrap set of notes, the simulation took the following amount of time:

> start.time <- proc.time()

> set.seed(7127)

> save.intervals <- t(apply(X = y.sim, MARGIN = 1, FUN

 = sim.func))

> 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: 13.56117

Note that this code was run on my Surface using a balanced power setting.

Next, I am going to use parallel processing to see if I can calculate the estimated true confidence levels faster. A key component of using this package is to put all of the code in one function that can be called by the function that does the actual parallel processing part. Below is my code copied directly from Tinn-R:

per.core1 <- function(X, alpha, R) {

 library(boot)

 calc.t2 <- function(data, i) {

 d2 <- data[i]

 var(d2)

 }

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

 d <- data[i]

 n <- length(d)

 l.jack <- empinf(data = d, statistic = calc.t2, stype

 = "i", type = "jack")

 v.jack <- var.linear(L = l.jack)

 c(var(d), v.jack)

 }

 sim.func <- function(y, alpha, R) {

 n <- length(y)

 t <- var(y)

 normal.based <- (n - 1)\*t / qchisq(p = c(1-alpha/2,

 alpha/2), df = n - 1)

 mu.hat4 <- 1/n\*sum((y - mean(y))^4)

 asym <- t + qnorm(p = c(alpha/2, 1-alpha/2)) \*

 sqrt((mu.hat4 - t^2)/n)

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

 sim = "ordinary")

 save.int <- boot.ci(boot.out = boot.res, conf = 1-

 alpha, type = "all")

 basic <- c(save.int$basic[4], save.int$basic[5])

 percentile <- c(save.int$perc[4], save.int$perc[5])

 bca <- c(save.int$bca[4], save.int$bca[5])

 student <- c(save.int$student[4],

 save.int$student[5])

 c(normal.based, asym, basic, percentile, bca,

 student)

 }

Here’s how I run the code using only the first four data sets:

> library(parallel)

> #TESTING

> start.time<-proc.time()

> cl <- makeCluster(2)

> clusterSetRNGStream(cl = cl, 8881)

> sim.intervals.PP <- parLapply(cl = cl, X =

 list(y.sim[1:2,], y.sim[3:4,]), fun = per.core1, alpha=

 0.05, R = 1999)

> save.all<-do.call(what = rbind, args = sim.intervals.PP)

> save.all

 [,1] [,2] [,3] [,4] [,5]

[1,] 1.8105389 14.564633 0.4607171 7.476027 0.3400175

[2,] 3.4175475 27.491994 3.4218801 11.559406 3.7492106

[3,] 0.4750438 3.821425 0.7009345 1.381485 0.5718528

[4,] 2.7335330 21.989533 -0.2934615 12.276277 0.1383994

 [,6] [,7] [,8] [,9] [,10]

[1,] 7.155083 0.7816615 7.596727 1.0965726 9.236297

[2,] 13.579298 1.4019878 11.232075 2.2531100 12.581648

[3,] 1.673889 0.4085307 1.510567 0.6560319 1.778607

[4,] 11.185726 0.7970891 11.844416 1.1667834 15.402443

 [,11] [,12]

[1,] -0.4550045 34.040759

[2,] 2.0949036 43.448855

[3,] 0.6202804 2.291554

[4,] -2.4834345 93.204286

> 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: 0.07083333

Comments about the code:

* The makeCluster() function tells R how many cores to use.
* The clusterSetRNGStream() function enables the use of separate seed numbers for each core.
* The parLapply() function works like the apply() and lapply() function. The “L” instructs R to return the results from each call to per.core1() in a list format. While not shown, the sim.intervals.PP object is actually a list of two components. Each component is a 25012 matrix. I combine the components into one matrix using the do.call() and rbind() functions.
* Again, all of the code needs to be put into the function called by parLapply(). This includes calls to library(). Also, items in R’s workspace are not available other than those that you put as arguments in parLapply().
* Below are additional comments about the arguments given in parLapply() :
	+ The cl argument specifies the number of cores for parallel processing. Thus, the per.core1() 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 = x, FUN = mean)

$a

[1] 5.5

$beta

[1] 4.535125

$logic

[1] 0.5

For our implementation of parLapply(), the X argument is our list that splits the simulated data into two parts for the two cores. Each part then is passed in as the first argument of per.core1().

* The … argument in parLapply() allows for additional items to be passed into the per.core1() function.

An alternative way to code the function called by parLapply() is to use an external file that contains most of the code. For example,

 per.core2 <- function(my.data, alpha, R) {

 setwd(dir = "C:\\chris")

 source(file = "Run\_on\_each\_core.R")

 save.int<-t(apply(X = my.data, MARGIN = 1, FUN =

 sim.func, alpha = alpha, R = R)) *#Oddly, this code*

 *will not work if it is in the source program*

 save.int

 }

where Run\_on\_each\_core.R contains the code that was in per.core1() before.

Below is my code and output from using all 500 simulated data sets:

> start.time<-proc.time()

> cl <- makeCluster(2)

> clusterSetRNGStream(cl = cl, 8881)

> sim.intervals.PP <- parLapply(cl = cl, X =

 list(y.sim[1:250,], y.sim[251:500,]), fun = per.core2,

 alpha = 0.05, R = 1999)

> save.all<-do.call(what = rbind, args = sim.intervals.PP)

> 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: 6.6425

Notice the amount of time here is about half the amount of time the MC simulations took in the bootstrap section of the notes!

Below is the summary of the intervals using my summarize() function:

> normal.based <- summarize(low.up = save.all[,1:2],

 sigma.sq = sigma^2)

> asym <- summarize(low.up = save.all[,3:4], sigma.sq =

 sigma^2)

> basic <- summarize(low.up = save.all[,5:6], sigma.sq =

 sigma^2)

> percentile <- summarize(low.up = save.all[,7:8], sigma.sq

 = sigma^2)

> bca <- summarize(low.up = save.all[,9:10], sigma.sq =

 sigma^2)

> student <- summarize(low.up = save.all[,11:12], sigma.sq

 = sigma^2)

> interval.names <- c("Normal", "Asymptotic", "Basic",

 "Percentile", "BCa", "Student")

> data.frame(interval = interval.names, rbind(normal.based,

 asym, basic, percentile, bca, student))

 interval true.conf exp.length exclude

1 Normal 0.960 15.802934 0

2 Asymptotic 0.712 5.792117 0

3 Basic 0.784 6.920417 0

4 Percentile 0.780 6.920417 0

5 BCa 0.806 7.972589 0

6 Student 0.946 30.591656 0

Notice that the bootstrap simulation results are not exactly the same as before due to the different seeds used with the resampling.

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

With Windows 8, the Task Manager gives information about the cores being used. Below is a screen capture while running the previous code on my Surface:



Because each thread is counted separately, the Task Manager indicates that approximately 50% of my processing capability is being used. Notice that the speed of the processor goes up to 2.35GHZ. This is how newer processors work – they increase their speed when needed and decrease when not needed.

With Windows 7, you can watch a Windows gadget that monitors your processor. For example, below is a screen capture taken while running the previous code on my Fujitsu.



Approximately 50% of my processing capability is being used. This gadget counts each thread separately.

The parallel processing is accomplished by R opening up additional R sessions. Because I wanted to use two cores as specified in makeCluster(), two additional sessions of R are opened. To see this, start the Task Manager and look at the number of R processes running. Below is a screen capture from running the previous code on my Surface.



Below is a screen capture from running the previous code on my Fujitsu.



I re-ran the same code, but changed the number of cores requested. I kept track of the time it took by core to do some benchmarking. Note that I had to change the value of the cl object and the X argument value in parLapply(). Below is a plot summarizing the time on my Surface:



We see that there was not much benefit from requesting more than two cores. Of course, when I request more than two cores, this causes multiple threads to be used for a single core.

Of course, when I used 4 cores (threads), there was no processing power available for other tasks. This was not as much of a problem on my Surface as it was on my Fujitsu where I practically could not do anything.

I performed the same benchmarking on my 6-core Dell desktop as well:



Note that the y-axis scale is different than in the previous plot. Overall, we see there are time savings for each additional core used.

When we first ran these simulations in the bootstrap section, we obtained warning messages due to problems with the BCa interval calculations. Unfortunately, these warning messages will not automatically be printed to the master R session because the warnings occur on the worker R sessions.

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

Revolution Analytics developed the foreach package for parallel processing. 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>.

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 a function for 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 by column. The list function puts the results into a list. Note that you need to specify just the function name rather than the function with parentheses like c().
* 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: Estimate true confidence level for a confidence interval (MC\_sim\_PP.R)

Below is how the simulation would be performed with a regular for loop:

> library(doParallel)

> cl<-makeCluster(spec = 2)

> registerDoParallel(cl = cl)

> start.time<-proc.time()

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

 streams of seeds

> save.all2<-foreach(i = 1:500, .combine = rbind, .packages

 = "boot") %dopar% {

 sim.func(y = y.sim[i,])

 }

> 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: 6.813

> normal.based <- summarize(low.up = save.all2[,1:2],

 sigma.sq = sigma^2)

> asym <- summarize(low.up = save.all2[,3:4], sigma.sq =

 sigma^2)

> basic <- summarize(low.up = save.all2[,5:6], sigma.sq =

 sigma^2)

> percentile <- summarize(low.up = save.all2[,7:8],

 sigma.sq = sigma^2)

> bca <- summarize(low.up = save.all2[,9:10], sigma.sq =

 sigma^2)

> student <- summarize(low.up = save.all2[,11:12], sigma.sq

 = sigma^2)

> interval.names <- c("Normal", "Asymptotic", "Basic",

 "Percentile", "BCa", "Student")

> data.frame(interval = interval.names, rbind(normal.based,

 asym, basic, percentile, bca, student))

 interval true.conf exp.length exclude

1 Normal 0.960 15.802934 0

2 Asymptotic 0.712 5.792117 0

3 Basic 0.790 6.919276 0

4 Percentile 0.782 6.919276 0

5 BCa 0.804 7.959627 0

6 Student 0.946 31.666950 0

This code took a little longer than with the parallel package.

Final comments

* Some users recommend foreach() only when there are a few combinations of items, because there is supposedly a lot of time spent with the combination of the results at the end.
* 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 may be slow while you use it! I usually leave one core open.
* The parallel package has other functions that may be useful such as parApply() and parRapply() which work like the apply() function.