Parallel computing in R

Robert Bagchi

March 22, 2016

1 Introduction

Many tasks we undertake in statistical computing involve a lot of repitition. Often each repeat is independent of the others - for example, we might want to make independent random draws from a distribution. These tasks are often described as "embarrassingly parallel" because we do not have to do them in sequence - they could all be done at the same time so a job that took (number of repeats) x (time of each repeat) could be reduced to time of each repeat, should we have sufficient resources. Most modern computers are capable of at least 4 parallel processes, which means you could accomplish tasks almost 4 times as fast!

Below is a simple example of a repetitive task.

```
> # let us do a simple calculation involving a lot
> ## of repetition, each task of which takes some time
>
> ## one common task is inverting a matrix - you
> ## have unconsciously done this many times when fitting
> ## linear models. Let us build a huge matrix (1000 x 1000)
> ## and invert it
> m <- matrix(runif(1e6), ncol=1e3, nrow=1e3)
> test <- solve(m) ## this takes some time
> ## to find out how long it takes, let us time it
> ## with system time
>
> system.time(test <- solve(m)) ## about 0.9 seconds

   user  system elapsed
   0.73    0.00    0.73

> ## now let us do this 20 times in sequence.
> ## I will use a for loop first.
>
> result <- list()
> ## this takes about 20 x the time
```

```
> system.time(for(i in 1:20){
+   m <- matrix(runif(1e6), ncol=1e3, nrow=1e3)
+   result[[i]] <- solve(m)
+ })

   user  system elapsed
  15.44    0.17   15.63

> ## we can sometimes speed up a little by using the apply
> ## functions.
> invertmatrix <- function(){
+   m <- matrix(runif(1e6), ncol=1e3, nrow=1e3)
+   return(solve(m))
+ }
> ## First write the function we want to repeat
> system.time(result <- sapply(1:20, function(i){
+   invertmatrix()
+ }, simplify=FALSE))

   user  system elapsed
  16.11    0.14   16.27

>
> ## but doesn't really make a difference here
```

2 Running parallel tasks on your computer

Parallel support is provided by the parallel package, which ships with base R. It includes a number of functions that mirror the apply series of functions, for example, sapply and lapply. The parallel versions of these functions are parSapply and parLapply.

```
> library(parallel)
> ## You can find the number of cores on your machine with
> detectCores()

[1] 8

> ## first set up  a cluster - asking for 6 cpus
> cl <- makeCluster(spec = 6)
> ## we then have to pass each of those nodes the function
> clusterExport(cl, varlist = 'invertmatrix')
> system.time(result <- parSapply(cl, 1:20, function(i){
+   invertmatrix()
+ }, simplify=FALSE))

   user  system elapsed
   0.23    0.40    7.38
```

```
> ## and it should complete the task in less time
>
> ## after you're done you should close the cluster
> stopCluster(cl)
```

The reason it's not 1/6 rd the time is there is some overhead when using parallel processing - the computer needs to pass data among different parts of the memory and so on. You will see a greater improvement if you do multiple iterations per cpu. The upshot is that the optimum number of CPUs is usually somewhat less than 1 CPU per iteration.

3 A few extensions and tips

Using parallel code is actually pretty simple. There are a few thigns taht it is worth knowing as you start getting set up. The first thing to know is that you not only have to export the objects from your working environment, like functions, but you also have to load libraries on each of the remote CPUs. Below is an example, using the mvrnorm function from the MASS package.

```
> ## If we don't have the package loaded, this doesn't work
> print(try(mvrnorm(5, mu=rep(0, 5), Sigma=diag(5))))

[1] "Error in try(mvrnorm(5, mu = rep(0, 5), Sigma = diag(5))) : \n  could not find functior
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in doTryCatch(return(expr), name, parentenv, handler): could not find function

> library(MASS)
> mvrnorm(5, mu=rep(0, 5), Sigma=diag(5)) ## now it does

           [,1]       [,2]        [,3]       [,4]        [,5]
[1,] -0.6419951 -2.2733075 -0.07894389 -1.8811210 -0.50284088
[2,] -0.3262736 -0.9581634  0.73190037 -0.1987236  0.03223303
[3,]  1.1743951  2.0697934 -1.32773185 -0.9053403  0.36125543
[4,] -0.9245566  0.4329050  0.42116413 -0.5994861 -1.72722346
[5,]  0.3582403 -0.3983205  0.12304497  0.2331361 -0.98510330

> cl <- makeCluster(spec = 3)
> ## we then have to pass each of those nodes the function
> try(parSapply(cl, 1:3, function(i) mvrnorm(5, mu=rep(0, 5), Sigma=diag(5))),
+     silent=FALSE)
> ## gives us the same error
> ## you can do the same function on every CPU by doing
> clusterEvalQ(cl, library(MASS)) ## which loads the library on every cpu
```

```
[[1]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[2]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[3]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"
```

```
> ## now
> parSapply(cl, 1:3, function(i) mvrnorm(5, mu=rep(0, 5), Sigma=diag(5)))
```

```
             [,1]        [,2]        [,3]
 [1,]  1.09461031 -0.67216195  2.23253053
 [2,] -0.41526310  1.24837365 -1.30185344
 [3,] -0.51719987  1.48272354 -0.43296224
 [4,]  0.35378169 -0.74395213 -0.84876294
 [5,] -0.96877912  0.24337718  1.19212861
 [6,]  2.00405110  2.10837443  0.74617093
 [7,]  0.76827193 -0.40699727 -0.59832497
 [8,] -0.45583475 -0.05037491  0.54637978
 [9,] -1.56362305 -1.18238844  1.05308145
[10,] -0.80403415 -0.46973095  0.25378466
[11,] -0.09064667 -2.14324608 -0.08642724
[12,] -0.21089665 -0.91783388 -0.48673538
[13,] -0.01800967  0.77145191  0.35289584
[14,]  1.00596416 -2.42161324 -0.91271536
[15,]  0.82637963  0.74500827 -0.83839445
[16,]  1.16377821  0.04255070  1.21214893
[17,] -0.68133138  0.83625492 -0.02809047
[18,] -1.25597350  1.73258992 -1.03542153
[19,]  2.39011423  1.40214713 -0.13017650
[20,]  1.67994089 -0.33377533  0.12953825
[21,] -0.34709979 -0.65509737  0.04175148
[22,] -0.12072020  1.04263730 -0.22038184
[23,] -0.91496792  0.49693519  0.18587944
[24,] -0.68608085  0.45337530 -0.14141812
[25,]  0.33220163  0.88860122 -1.07224935
```

```
> ## works
>
> ## This does produce a bit of a mess because sapply tries to simplify your result
> ## by default. To fix this we can do
```

```
> parSapply(cl, 1:3, function(i) mvrnorm(5, mu=rep(0, 5), Sigma=diag(5)),
+           simplify=FALSE)

[[1]]
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,]  0.4250010  1.5335188  0.3506629 -0.9038917  0.1644789
[2,]  0.9478573 -0.2106198 -1.5982024 -1.5675257  0.4366550
[3,] -0.2052678 -1.0674884  0.3690451 -2.4427365  0.6900964
[4,] -0.8967369 -0.2415691 -0.3025350  1.5988644 -0.9830172
[5,] -0.7789410 -1.5579440 -0.5712570  0.5594091  0.9368609

[[2]]
           [,1]       [,2]       [,3]       [,4]         [,5]
[1,] -0.9091831  0.6605363  0.2530331  2.2645328  2.112650787
[2,]  0.6287691  2.0509536 -0.5983336  1.1258005  0.008447817
[3,]  1.1531525  0.7151855  0.7224122 -0.4301338 -0.547390555
[4,] -0.4877567  0.6111650  1.7199827  0.8331966  0.657299730
[5,]  0.2840941 -1.1698040  1.1713551  0.5109483 -0.766867418

[[3]]
           [,1]       [,2]       [,3]        [,4]       [,5]
[1,]  0.6815600 -0.1460675  0.9677285 -0.09759881 -1.7162018
[2,] -1.6941524 -1.9502681 -0.2798352 -0.07869260 -2.6478184
[3,] -0.5200145  0.5954674 -1.1244547 -0.42277572 -1.2736764
[4,]  0.7254521  2.1254104  0.1750677  0.89642724  0.1407217
[5,] -0.2757563  0.2131980  0.6039209 -0.39804847 -1.2541405

> ## Which I prefer
> ## or also use parLapply
> parLapply(cl, 1:3, function(i) mvrnorm(5, mu=rep(0, 5), Sigma=diag(5)))

[[1]]
            [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.61569681  0.5210132  0.3550141 -1.6424792  1.3132141
[2,] -0.07336971 -1.5993443 -0.2864664 -1.1146244 -1.4715848
[3,]  0.27835922  1.3520664 -0.3000576 -0.5791113  1.1710378
[4,] -0.45732703 -0.5853037 -1.1890823 -1.9062634 -1.0632187
[5,] -0.66581895  0.3544216 -0.5060245  0.7473242  0.8314192

[[2]]
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.2443090 -0.1400825 -0.1951302  0.4239697  2.8238367
[2,]  0.2386100  0.3338066  0.5393717  1.0137684  1.1273469
[3,] -0.6127080  2.0090164  0.3624080 -0.3064269  0.4310554
[4,]  0.9780489 -0.1954418  0.4039148  0.4442992 -0.2312604
[5,]  0.2786719 -0.8515629 -0.2281151 -0.3769463  1.0089378
```

```
[[3]]
            [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.11156884  0.4911012 -0.2726498 -0.0765165 -0.4104144
[2,] -0.68541380  0.4143387 -1.3708349  0.9440515  0.4873053
[3,]  1.15411025 -0.3559932 -1.2894510  0.1128527  0.7762199
[4,] -0.80678636  1.6401359 -0.4656340  0.7388628  0.9623169
[5,] -0.03324654 -0.7417401  0.4522963  1.3040484 -0.6703434

> ## Which is an alternative apply type function
>
> stopCluster(cl)
>
```

The other very useful thing to know is how to deal with multiple arguments. For example, in all the cases here, we only had one input that we were handing to the function to be run on each CPU (indeed, we actually had no arguments, and just had an index). The function in the apply family for multiple arguments is mapply, and the parallel version is clusterMap. This is how we use it

```
> ## lets continue with the mvrnorm example, but this time let the mean and variance
> ##  be different on different clusters
>
> test.mu <- list(rep(1, 5), rep(0, 5), rep(-1, 5))
> test.sigma <- list(diag(5), diag(5)/2, diag(5)*2)
> cl <- makeCluster(spec = 3)
> clusterEvalQ(cl, library(MASS)) ## repeat each time you set up a new cluster

[[1]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[2]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"

[[3]]
[1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "base"

> clusterMap(cl=cl, fun=mvrnorm, mu=test.mu, Sigma=test.sigma,
+            MoreArgs=list(n=5), SIMPLIFY=FALSE)

[[1]]
           [,1]      [,2]        [,3]        [,4]      [,5]
[1,]  1.1462855 2.5955663  1.24550707  1.14841223 1.1636479
[2,] -0.1729008 2.2362688 -0.63125845 -0.00601749 2.8472470
[3,] -0.3989909 1.0967590  0.89529308  2.34424789 0.1315231
```

```
[4,] -0.7487908 3.4608764  0.63111172  1.73565356 0.9221885
[5,] -1.3742869 0.3268554 -0.04645696  0.89431576 2.5941942

[[2]]
           [,1]       [,2]      [,3]       [,4]       [,5]
[1,]  0.6659752  0.3514910 0.4716962  0.3214650 -0.4115412
[2,]  0.1147094 -0.3266376 0.2932683  0.6943290  0.6048038
[3,] -1.4328098 -0.3431884 0.6192202 -0.2902896  1.0022569
[4,] -1.0412336 -0.3168193 0.4932622 -0.2118005  0.3459006
[5,] -0.3022864  0.6809992 1.1493617 -0.2200643 -0.7552925

[[3]]
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -2.5571234 -0.8982493 -0.7968208 -2.2058171  0.4173885
[2,] -3.5832580 -1.2325359  2.5206771 -1.7935697  0.7346477
[3,]  0.5223728 -0.5235832 -2.1950731 -1.7780063 -0.2827496
[4,] -1.3446998 -0.6378127 -1.6625299  0.5035915  1.8649682
[5,]  1.9888450 -1.9272290  1.9325310  0.3402891 -0.4324658

> 
> ## Things that don't vary are handed as a named list called MoreArgs.
> ## Things that vary should be handed over as lists, and all
> ## have to be of the same length.
```

4 Randomization

One *important* detail when running tasks that involve randomization is making sure that your random number generator is set up okay. This is particularly important if you are setting the random number seed for your analyses to be repeatable. Here is a quote from the vignette from the Parallel package:

> When an R process is started up it takes the random-number seed from the object .Random.seed in a saved workspace or constructs one from the clock time and process ID when random-number generation is first used (see the help on RNG). Thus worker processes might get the same seed because a workspace containing .Random.seed was restored or the random number generator has been used before forking: otherwise these get a non-reproducible seed (but with very high probability a different seed for each worker).

Here is an example of how it can all go wrong.

```
> ##rMean <- function(n) mean(rnorm(n))
> set.seed(123)
> sapply(1:5, function(i) rnorm(3))
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.5604756 0.07050839  0.4609162 -0.4456620  0.4007715
[2,] -0.2301775 0.12928774 -1.2650612  1.2240818  0.1106827
[3,]  1.5587083 1.71506499 -0.6868529  0.3598138 -0.5558411

> ## if we repeat this, we get the same result
> set.seed(123)
> sapply(1:5, function(i) rnorm(3))

           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.5604756 0.07050839  0.4609162 -0.4456620  0.4007715
[2,] -0.2301775 0.12928774 -1.2650612  1.2240818  0.1106827
[3,]  1.5587083 1.71506499 -0.6868529  0.3598138 -0.5558411

> set.seed(123)
> cl <- makeCluster(spec = 5)
> parSapply(cl = cl, 1:5, FUN=function(i) rnorm(3))

           [,1]       [,2]        [,3]      [,4]      [,5]
[1,] -0.5875195  0.3288642  0.02299133 1.8362055 0.7332513
[2,] -0.5108612  0.2975473 -0.86918611 0.6980254 0.4661642
[3,]  2.5362440 -1.7049674  1.39930442 0.9155103 0.1826455

> stopCluster(cl)
> ## different results
> set.seed(123)
> cl <- makeCluster(spec = 5)
> ##clusterExport(cl, 'rMean')
> ##parSapply(cl = cl, 1:5, FUN=function(i) rMean(n=100))
> parSapply(cl = cl, 1:5, FUN=function(i) rnorm(3))

            [,1]       [,2]      [,3]      [,4]       [,5]
[1,]  2.08553037 -0.4781369 0.7627636 0.4845697 -0.7272328
[2,]  1.11754585  1.3121917 1.5237677 0.9735818 -1.5746030
[3,] -0.09388983 -0.5181009 1.7004961 1.3154911 -1.1433474

> stopCluster(cl)
> ## And the results are unrepeatable.
>
> ## The wrong solution is to put set seed in the function
> cl <- makeCluster(spec = 5)
> parSapply(cl = cl, 1:5, FUN=function(i) {
+   set.seed(123)
+   rnorm(3)
+ })

           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -0.5604756 -0.5604756 -0.5604756 -0.5604756 -0.5604756
[2,] -0.2301775 -0.2301775 -0.2301775 -0.2301775 -0.2301775
[3,]  1.5587083  1.5587083  1.5587083  1.5587083  1.5587083
```

```
> stopCluster(cl) ## Not good - same numbers each time
```

So the results are different from the non-parallel analysis and also not repeatable among different runs. Perhaps that is not a problem, but given that the random seed is basically being set haphazardly on each CPU, and the exact way in which it is set is dependent on the options set on that machine, you *could* run into some very serious problems if you just steam ahead. The solution is to set up a random number stream that is shared by the different CPUs. The worst thing you can do is to set up your code so you set the seed within each parallel call. You basically end up with no variation.

We first need to change the kind of random number generator. I don't know why, but the default, Mersenne-Twister doesn't work so well in parallel processes.

```
> ## First set the kind of random number generator
> RNGkind("L'Ecuyer-CMRG")
> set.seed(123)
> sapply(1:5, function(i) rnorm(3))
```

```
           [,1]       [,2]       [,3]        [,4]       [,5]
[1,] -0.9685927 -1.8150926  0.1571934  0.00395328  0.1762544
[2,]  0.7061091  0.3304096 -2.0654072 -0.53711605 -1.5836803
[3,]  1.4890213 -1.1421557 -0.4405469 -0.01260453  0.4677918
```

```
> ##We get different results from the previous chunk as
> ## we are using a different RNG
>
> set.seed(123)
> sapply(1:5, function(i) rnorm(3))
```

```
           [,1]       [,2]       [,3]        [,4]       [,5]
[1,] -0.9685927 -1.8150926  0.1571934  0.00395328  0.1762544
[2,]  0.7061091  0.3304096 -2.0654072 -0.53711605 -1.5836803
[3,]  1.4890213 -1.1421557 -0.4405469 -0.01260453  0.4677918
```

```
> ## But repeatable differences
> cl <- makeCluster(spec=5)
> clusterSetRNGStream(cl = cl, iseed = 123)
> parSapply(cl = cl, 1:5, FUN=function(i) rnorm(3))
```

```
           [,1]       [,2]        [,3]      [,4]      [,5]
[1,] -0.9685927 -0.4094454 -0.48906078 -1.038866 0.7613014
[2,]  0.7061091  0.8909694  0.43304237  1.574512 2.2994158
[3,]  1.4890213 -0.8653704 -0.03195349  0.747082 0.2002062
```

```
> stopCluster(cl)
> ## Which are a different but
> cl <- makeCluster(spec=5)
> clusterSetRNGStream(cl = cl, iseed = 123)
> parSapply(cl = cl, 1:5, FUN=function(i) rnorm(3))
```

```
           [,1]       [,2]        [,3]      [,4]      [,5]
[1,] -0.9685927 -0.4094454 -0.48906078 -1.038866 0.7613014
[2,]  0.7061091  0.8909694  0.43304237  1.574512 2.2994158
[3,]  1.4890213 -0.8653704 -0.03195349  0.747082 0.2002062

> stopCluster(cl)
> ## The same as another instance.
>
> ## Note that this is dependent of the number of cppus
> cl <- makeCluster(spec=3)
> clusterSetRNGStream(cl = cl, iseed = 123)
> parSapply(cl = cl, 1:5, FUN=function(i) rnorm(3))

           [,1]       [,2]       [,3]        [,4]       [,5]
[1,] -0.9685927 -1.8150926 -0.4094454 -0.48906078  0.1467037
[2,]  0.7061091  0.3304096  0.8909694  0.43304237 -1.7523909
[3,]  1.4890213 -1.1421557 -0.8653704 -0.03195349 -1.0851006

> stopCluster(cl)
>
> ## Gives you different numbers
```