Random Number Generator for Parallel Computing in R

Last updated on Jul 22, 2019 · 2 min read · 0 Comments

There are several ways to generate random numbers in R package "parallel". Here I used **mcmapply** as an example to show how to generate 5 random numbers from 4 cores.

Same seed for each core

When you set *MC.set.seed* as FALSE, each core has the same seed and generate the same random numbers.

```
set.seed(123)
mcmapply(rnorm, n=rep(5,4), mc.set.seed = FALSE, mc.cores = 4)
```

As shown below, 4 cores generate 4 series (4 columns) of random numbers. The 4 columns are the same because they have same seed.

```
##           [,1]       [,2]       [,3]       [,4]
## [1,] -0.9685927 -0.9685927 -0.9685927 -0.9685927
## [2,]  0.7061091  0.7061091  0.7061091  0.7061091
## [3,]  1.4890213  1.4890213  1.4890213  1.4890213
## [4,] -1.8150926 -1.8150926 -1.8150926 -1.8150926
## [5,]  0.3304096  0.3304096  0.3304096  0.3304096
```

Different seeds for each core

When you set *MC.set.seed* as TRUE, each core has different seeds and will generate different random numbers.

Without reproducibility

There are several methods in R to generate random numbers, the default method is "Mersenne-Twister". With this method, we cannot get the same random numbers with the same seed.

```
RNGkind("Mersenne-Twister")
set.seed(123)
mcmapply(rnorm, n=rep(5,4), mc.set.seed = TRUE, mc.cores = 4)
```

For example, we run the code above and get the following random numbers.

```
##           [,1]       [,2]        [,3]        [,4]
## [1,] -0.2261798 -0.6251189 -2.37868583  1.20695891
## [2,]  2.4251365  0.3667103  1.54679159 -0.09272884
## [3,]  0.7750225 -1.5508046 -0.56366992 -0.46451586
## [4,]  0.0124790 -1.5446089 -0.62513942 -0.73186601
## [5,]  0.8173900 -1.2199431  0.07999375  0.96084955
```

The columns are different because each core has different seeds. If we run the same code again, we will get different random numbers even we still set the same seed:

```
##           [,1]      [,2]       [,3]        [,4]
## [1,]  2.2683726 0.1390017 -1.0274918 -0.28896864
## [2,] -0.4140674 2.1043673 -2.7126309 -0.62731623
## [3,] -0.4327392 0.8579146  0.8667401  0.24681518
## [4,]  0.3623787 0.9685905  0.2008967  0.04581639
## [5,]  1.3556560 0.4316045  0.3484889 -0.16804107
```

With reproducibility

To solve the reproducibility problem, we need to set the random number generator method to "L'Ecuyer-CMRG" and use the following code.

```
RNGkind("L'Ecuyer-CMRG")
set.seed(123)
mc.reset.stream()
mcmapply(rnorm, n=rep(5,4), mc.set.seed = TRUE, mc.cores = 4)
```

Run for the fisrt time:

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] -0.4094454 -0.48906078 -1.0388664  0.7613014
## [2,]  0.8909694  0.43304237  1.5745125  2.2994158
## [3,] -0.8653704 -0.03195349  0.7470820  0.2002062
## [4,]  1.4642711  0.14670372  0.6718720 -0.2975786
## [5,]  1.2674845 -1.75239095  0.2691436  0.4608767
```

Run for the second time:

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] -0.4094454 -0.48906078 -1.0388664  0.7613014
## [2,]  0.8909694  0.43304237  1.5745125  2.2994158
## [3,] -0.8653704 -0.03195349  0.7470820  0.2002062
## [4,]  1.4642711  0.14670372  0.6718720 -0.2975786
## [5,]  1.2674845 -1.75239095  0.2691436  0.4608767
```

The same!

Gang Peng

Associate Research Scientist in Biostatistics

My research interests include biostatistics, bioinformatics, and data mining.

cv