# Using the `doRNG` package

*doRNG* package – Version 1.7.1

Renaud Gaujoux

June 18, 2018

## Contents

## Introduction

Research reproducibility is an issue of concern, e.g. in bioinformatics [**Hothorn2011**, **Stodden2011**, **Ioannidis2008**]. Some analyses require multiple independent runs to be performed, or are amenable to a split-and-reduce scheme. For example, some optimisation algorithms are run multiple times from different random starting points, and the result that achieves the least approximation error is selected. The *foreach* package[1] [**Rpackage:foreach**] provides a very convenient way to perform parallel computations, with different parallel environments such as MPI or Redis, using a transparent loop-like syntax:

[1] `https://cran.r-project.org/package=foreach`

```
# load and register parallel backend for multicore computations
library(doParallel)

## Loading required package:  foreach
## Loading required package:  iterators
## Loading required package:  parallel

cl <- makeCluster(2)
registerDoParallel(cl)

# perform 5 tasks in parallel
x <- foreach(i=1:5) %dopar% {
        i + runif(1)
}
unlist(x)

## [1] 1.336610 2.894867 3.414720 4.905941 5.414722
```

For each parallel environment a *backend* is implemented as a specialised `%dopar%` operator, which performs the setup and pre/post-processing specifically required by the environment (e.g. export of variable to each worker). The `foreach` function and the `%dopar%` operator handle the generic parameter dispatch when the task are split between worker processes, as well as the reduce step – when the results are returned to the master worker.

When stochastic computations are involved, special random number generators must be used to ensure that the separate computations are indeed statistically independent – unless otherwise wanted – and that the loop is reproducible. In particular, standard `%dopar%` loops are not reproducible:

```
# with standard %dopar%: foreach loops are not reproducible
set.seed(123)
res <- foreach(i=1:5) %dopar% { runif(3) }
set.seed(123)
res2 <- foreach(i=1:5) %dopar% { runif(3) }
identical(res, res2)

## [1] FALSE
```

A random number generator commonly used to achieve reproducibility is the combined multiple-recursive generator from **Lecuyer1999** [**Lecuyer1999**]. This generator can generate independent random streams, from a 6-length numeric seed. The idea is then to generate a sequence of random stream of the same length as the number of iteration (i.e. tasks) and use a different stream when computing each one of them.

The *doRNG* package[2] [**Rpackage:doRNG**] provides convenient ways to implement reproducible parallel `foreach` loops, independently of the parallel backend used to perform the computation. We illustrate its use, showing how non-reproducible loops can be made reproducible, even when tasks are not scheduled in the same way in two separate set of

[2]https://cran.r-project.org/package=doRNG

runs, e.g. when the workers do not get to compute the same number of tasks or the number of workers is different. The package has been tested with the *doParallel*[3] and *doMPI*[4] packages [**Rpackage:doMPI**, **Rpackage:doParallel**], but should work with other backends such as provided by the *doRedis* package[5] [**Rpackage:doRedis**].

## 1 The `%dorng%` operator

The *doRNG* package defines a new generic operator, `%dorng%`, to be used with foreach loops, instead of the standard %dopar%. Loops that use this operator are *de facto* reproducible.

```
# load the doRNG package
library(doRNG)

## Loading required package:  rngtools

# using %dorng%: loops _are_ reproducible
set.seed(123)
res <- foreach(i=1:5) %dorng% { runif(3) }
set.seed(123)
res2 <- foreach(i=1:5) %dorng% { runif(3) }
identical(res, res2)

## [1] TRUE
```

### 1.1 How it works

For a loop with $N$ iterations, the `%dorng%` operator internally performs the following tasks:

1. generate a sequence of random seeds $(S_i)_{1\leq i\leq N}$ for the $R$ random number generator `"L'Ecuyer-CMRG"` [**Lecuyer1999**], using the function `nextRNGStream` from the *parallel* package[6] [**Rpackage:parallel**], which ensure the different RNG streams are statistically independent;

2. modify the loop's $R$ expression so that the random number generator is set to `"L'Ecuyer-CMRG"` at the beginning of each iteration, and is seeded with consecutive seeds in $(S_n)$: iteration $i$ is seeded with $S_i$, $1 \leq i \leq N$;

3. call the standard `%dopar%` operator, which in turn calls the relevant (i.e. registered) foreach parallel backend;

[3] https://cran.r-project.org/package=doParallel
[4] https://cran.r-project.org/package=doMPI
[5] https://cran.r-project.org/package=doRedis
[6] https://cran.r-project.org/package=parallel

4. store the whole sequence of random seeds as an attribute in the result object:

```
attr(res, 'rng')
```

```
## [[1]]
## [1]         407   642048078    81368183 -2093158836   506506973  1421492218 -1906381517
##
## [[2]]
## [1]         407  1340772676 -1389246211  -999053355  -953732024  1888105061  2010658538
##
## [[3]]
## [1]         407 -1318496690  -948316584   683309249  -990823268 -1895972179  1275914972
##
## [[4]]
## [1]         407   524763474  1715794407  1887051490 -1833874283   494155061 -1221391662
##
## [[5]]
## [1]         407 -1816009034  -580124020  1603250023   817712173   190009158  -706984535
```

## 1.2 Seeding computations

Sequences of random streams for "L'Ecuyer-CMRG" are generated using a 6-length integer seed, e.g.,:

```
nextRNGStream(c(407L, 1:6))
```

```
## [1]         407  -447371532   542750874  -935969228  -269326340   701604884 -1748056907
```

However, the %dorng% operator provides alternative – convenient – ways of seeding reproducible loops.

**set.seed:** as shown above, calling set.seed before the loop ensure reproducibility of the results, using a single integer as a seed. The actual 6-length seed is then generated with an internal call to RNGkind("L'Ecuyer-CMRG").

**.options.RNG with single integer:** the %dorng% operator support options that can be passed in the foreach statement, containing arguments for the internal call to set.seed:

```
# use a single numeric as a seed
s <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
s2 <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
identical(s, s2)
```

```
## [1] TRUE
```

**Note**: calling `set.seed` before the loop is equivalent to passing the seed in `.options.RNG`. See Section 1.3 for more details.

The kind of Normal generator may also be passed in `.options.RNG`:

```
## Pass the Normal RNG kind to use within the loop
# results are identical if not using the Normal kind in the loop
optsN <- list(123, normal.kind="Ahrens")
resN.U <- foreach(i=1:5, .options.RNG=optsN) %dorng% { runif(3) }
identical(resN.U[1:5], res[1:5])

## [1] TRUE

# Results are different if the Normal kind is used and is not the same
resN <- foreach(i=1:5, .options.RNG=123) %dorng% { rnorm(3) }
resN1 <- foreach(i=1:5, .options.RNG=optsN) %dorng% { rnorm(3) }
resN2 <- foreach(i=1:5, .options.RNG=optsN) %dorng% { rnorm(3) }
identical(resN[1:5], resN1[1:5])

## [1] FALSE

identical(resN1[1:5], resN2[1:5])

## [1] TRUE
```

**`.options.RNG` with 6-length:** the actual 6-length integer seed used for the first RNG stream may be passed via `options.RNG`:

```
# use a 6-length numeric
s <- foreach(i=1:5, .options.RNG=1:6) %dorng% { runif(3) }
attr(s, 'rng')[1:3]

## [[1]]
## [1] 407   1   2   3   4   5   6
##
## [[2]]
## [1]         407  -447371532   542750874  -935969228  -269326340   701604884 -1748056907
##
## [[3]]
## [1]         407   311773008 -1393648596   433058656  -545474683  2059732357   994549473
```

**`.options.RNG` with 7-length:** a 7-length integer seed may also be passed via `options.RNG`, which is useful to seed a loop with the value of `.Random.seed` as used in some iteration of another loop[7]:

[7]Note that the RNG kind is then always required to be the `"L'Ecuyer-CMRG"`, i.e. the first element of the seed must have unit 7 (e.g. 407 or 107).

```
# use a 7-length numeric, used as first value for .Random.seed
seed <- attr(res, 'rng')[[2]]
s <- foreach(i=1:5, .options.RNG=seed) %dorng% { runif(3) }
identical(s[1:4], res[2:5])

## [1] TRUE
```

**`.options.RNG` with complete sequence of seeds:** the complete description of the sequence of seeds to be used may be passed via `options.RNG`, as a list or a matrix with the seeds in columns. This is useful to seed a loop exactly as desired, e.g. using an RNG other than `"L'Ecuyer-CMRG"`, or using different RNG kinds in each iteration, which probably have different seed length, in order to compare their stochastic properties. It also allows to reproduce `%dorng%` loops without knowing their seeding details:

```
# reproduce previous %dorng% loop
s <- foreach(i=1:5, .options.RNG=res) %dorng% { runif(3) }
identical(s, res)

## [1] TRUE

## use completely custom sequence of seeds (e.g. using RNG "Marsaglia-Multicarry")
# as a matrix
seedM <- rbind(rep(401, 5), mapply(rep, 1:5, 2))
seedM

##      [,1] [,2] [,3] [,4] [,5]
## [1,]  401  401  401  401  401
## [2,]    1    2    3    4    5
## [3,]    1    2    3    4    5

sM <- foreach(i=1:5, .options.RNG=seedM) %dorng% { runif(3) }
# same seeds passed as a list
seedL <- lapply(seq(ncol(seedM)), function(i) seedM[,i])
sL <- foreach(i=1:5, .options.RNG=seedL) %dorng% { runif(3) }
identical(sL, sM)

## [1] TRUE
```

## 1.3 Difference between `set.seed` and `.options.RNG`

While it is equivalent to seed `%dorng%` loops with `set.seed` and `.options.RNG`, it is important to note that the result depends on the current RNG kind [8]:

[8]See Section 7 about a bug in versions ¡ 1.4 on this feature.

```
# default RNG kind
RNGkind('default')
def <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }

# Marsaglia-Multicarry
RNGkind('Marsaglia')
mars <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
identical(def, mars)

## [1] FALSE

# revert to default RNG kind
RNGkind('default')
```

This is a "normal" behaviour, which is a side-effect of the expected equivalence between `set.seed` and `.options.RNG`. This should not be a problem for reproducibility though, as R RNGs are stable across versions, and loops are most of the time used with the default RNG settings. In order to ensure seeding is independent from the current RNG, one has to pass a 7-length numeric seed to `.options.RNG`, which is then used directly as a value for `.Random.seed` (see below).

## 2 Parallel environment independence

An important feature of `%dorng%` loops is that their result is independent of the underlying parallel physical settings. Two separate runs seeded with the same value will always produce the same results. Whether they use the same number of worker processes, parallel backend or task scheduling does not influence the final result. This also applies to computations performed sequentially with the `doSEQ` backend. The following code illustrates this feature using 2 or 3 workers.

```
# define a stochastic task to perform
task <- function() c(pid=Sys.getpid(), val=runif(1))

# using the previously registered cluster with 2 workers
set.seed(123)
res_2workers <- foreach(i=1:5, .combine=rbind) %dorng% {
        task()
}
# stop cluster
stopCluster(cl)

# Sequential computation
registerDoSEQ()
set.seed(123)
res_seq <- foreach(i=1:5, .combine=rbind) %dorng% {
        task()
}
```

```
#

# Using 3 workers
# NB: if re-running this vignette you should edit to force using 3 here
cl <- makeCluster( if(isManualVignette()) 3 else 2)
length(cl)

## [1] 2

# register new cluster
registerDoParallel(cl)
set.seed(123)
res_3workers <- foreach(i=1:5, .combine=rbind) %dorng% {
        task()
}
# task schedule is different
pid <- rbind(res1=res_seq[,1], res_2workers[,1], res2=res_3workers[,1])
storage.mode(pid) <- 'integer'
pid

##      result.1 result.2 result.3 result.4 result.5
## res1    10446    10446    10446    10446    10446
##         10461    10473    10461    10473    10461
## res2    10490    10502    10490    10502    10490

# results are identical
identical(res_seq[,2], res_2workers[,2]) && identical(res_2workers[,2], res_3workers[,2])

## [1] TRUE
```

## 3 Reproducible %dopar% loops

The *doRNG* package also provides a non-invasive way to convert `%dopar%` loops into reproducible loops, i.e. without changing their actual definition. It is useful to quickly ensure the reproducibility of existing code or functions whose definition is not accessible (e.g. from other packages). This is achieved by registering the `doRNG` backend:

```
set.seed(123)
res <- foreach(i=1:5) %dorng% { runif(3) }

registerDoRNG(123)
res_dopar <- foreach(i=1:5) %dopar% { runif(3) }
identical(res_dopar, res)

## [1] TRUE

attr(res_dopar, 'rng')
```

```
## [[1]]
## [1]         407   642048078    81368183 -2093158836   506506973  1421492218 -1906381517
##
## [[2]]
## [1]         407  1340772676 -1389246211  -999053355  -953732024  1888105061  2010658538
##
## [[3]]
## [1]         407 -1318496690  -948316584   683309249  -990823268 -1895972179  1275914972
##
## [[4]]
## [1]         407   524763474  1715794407  1887051490 -1833874283   494155061 -1221391662
##
## [[5]]
## [1]         407 -1816009034  -580124020  1603250023   817712173   190009158  -706984535
```

## 4 Reproducibile sets of loops

Sequences of multiple loops are reproducible, whether using the `%dorng%` operator or the registered `doRNG` backend:

```
set.seed(456)
s1 <- foreach(i=1:5) %dorng% { runif(3) }
s2 <- foreach(i=1:5) %dorng% { runif(3) }
# the two loops do not use the same streams: different results
identical(s1, s2)

## [1] FALSE

# but the sequence of loops is reproducible as a whole
set.seed(456)
r1 <- foreach(i=1:5) %dorng% { runif(3) }
r2 <- foreach(i=1:5) %dorng% { runif(3) }
identical(r1, s1) && identical(r2, s2)

## [1] TRUE

# one can equivalently register the doRNG backend and use %dopar%
registerDoRNG(456)
r1 <- foreach(i=1:5) %dopar% { runif(3) }
r2 <- foreach(i=1:5) %dopar% { runif(3) }
identical(r1, s1) && identical(r2, s2)

## [1] TRUE
```

## 5 Nested and conditional loops

Nested and conditional foreach loops are currently not supported and generate an error:

```
# nested loop
try( foreach(i=1:10) %:% foreach(j=1:i) %dorng% { rnorm(1) } )

## Error:  nested/conditional foreach loops are not supported yet.
## See the package's vignette for a work around.

# conditional loop
try( foreach(i=1:10) %:% when(i %% 2 == 0) %dorng% { rnorm(1) } )

## Error:  nested/conditional foreach loops are not supported yet.
## See the package's vignette for a work around.
```

In this section, we propose a general work around for this kind of loops, that will eventually be incorporated in the `%dorng%` operator – when I find out how to mimic its behaviour from the operator itself.

### 5.1 Nested loops

The idea is to create a sequence of RNG seeds before the outer loop, and use each of them successively to set the RNG in the inner loop – which is exactly what `%dorng%` does for simple loops:

```
# doRNG must not be registered
registerDoParallel(cl)

# generate sequence of seeds of length the number of computations
n <- 10; p <- 5
rng <- RNGseq( n * p, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% foreach(j=1:p, r=rng[(i-1)*p + 1:p]) %dopar% {

        # set RNG seed
    rngtools::setRNG(r)

    # do your own computation ...
    c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
res2 <- foreach(i=1:n) %:% foreach(j=1:p) %do%{
    # set seed
        rngtools::setRNG(rng[[k]])
        k <- k + 1
    # do your own computation ...
        c(i, j, rnorm(1))
}

stopifnot( identical(res, res2) )
```

The following is a more complex example with unequal – but **known *a priori*** – numbers of iterations performed in the inner loops:

```
# generate sequence of seeds of length the number of computations
n <- 10
rng <- RNGseq( n * (n+1) / 2, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% foreach(j=1:i, r=rng[(i-1)*i/2 + 1:i]) %dopar%{

        # set RNG seed
        rngtools::setRNG(r)

        # do your own computation ...
        c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
res2 <- foreach(i=1:n) %:% foreach(j=1:i) %do%{
        # set seed
        rngtools::setRNG(rng[[k]])
        k <- k + 1

        # do your own computation ...
        c(i, j, rnorm(1))
}

stopifnot( identical(res, res2) )
```

## 5.2 Conditional loops

The work around used for nested loops applies to conditional loops that use the `when()` clause. It ensures that the RNG seed use for a given inner iteration does not depend on the filter, but only on its index in the unconditional-unfolded loop:

```
# un-conditional single loop
resAll <- foreach(i=1:n, .options.RNG=1234) %dorng%{
        # do your own computation ...
        c(i, rnorm(1))
}
```

```
# generate sequence of RNG
rng <- RNGseq(n, 1234)

# conditional loop: even iterations
resEven <- foreach(i=1:n, r=rng) %:% when(i %% 2 == 0) %dopar%{

        # set RNG seed
        rngtools::setRNG(r)

        # do your own computation ...
        c(i, rnorm(1))
}

# conditional loop: odd iterations
resOdd <- foreach(i=1:n, r=rng) %:% when(i %% 2 == 1) %dopar%{

        # set RNG seed
        rngtools::setRNG(r)

        # do your own computation ...
        c(i, rnorm(1))
}

# conditional loop: only first 2 and last 2
resFL <- foreach(i=1:n, r=rng) %:% when(i %in% c(1,2,n-1,n)) %dopar%{

        # set RNG seed
        rngtools::setRNG(r)

        # do your own computation ...
        c(i, rnorm(1))
}

# compare results
stopifnot( identical(resAll[seq(2,n,by=2)], resEven) )
stopifnot( identical(resAll[seq(1,n,by=2)], resOdd) )
stopifnot( identical(resAll[c(1,2,n-1,n)], resFL) )
```

## 5.3 Nested conditional loops

Conditional nested loops may use the same work around, as shown in this intricate example:

```
# generate sequence of seeds of length the number of computations
n <- 10
rng <- RNGseq( n * (n+1) / 2, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% when(i %% 2 == 0) %:% foreach(j=1:i, r=rng[(i-1)*i/2 + 1:i]) %dopar%{
        # set RNG seed
        rngtools::setRNG(r)

        # do your own computation ...
        c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
resAll <- foreach(i=1:n) %:% foreach(j=1:i) %do%{
        # set seed
        rngtools::setRNG(rng[[k]])
        k <- k + 1

        # do your own computation ...
        c(i, j, rnorm(1))
}

stopifnot( identical(resAll[seq(2,n,by=2)], res) )
```

## 6 Performance overhead

The extra setup performed by the %dorng% operator leads to a slight performance overhead, which might be significant for very quick computations, but should not be a problem for realistic computations. The benchmarks below show that a %dorng% loop may take up to two seconds more than the equivalent %dopar% loop, which is not significant in practice, where parallelised computations typically take several minutes.

```
# load rbenchmark
library(rbenchmark)

# comparison is done on sequential computations
registerDoSEQ()
rPar <- function(n, s=0){ foreach(i=1:n) %dopar% { Sys.sleep(s) } }
rRNG <- function(n, s=0){ foreach(i=1:n) %dorng% { Sys.sleep(s) } }

# run benchmark
cmp <- benchmark(rPar(10), rRNG(10)
                        , rPar(25), rRNG(25)
                        , rPar(50), rRNG(50)
                        , rPar(50, .01), rRNG(50, .01)
            , rPar(10, .05), rRNG(10, .05)
                        , replications=5)
# order by increasing elapsed time
cmp[order(cmp$elapsed), ]

##               test replications elapsed relative user.self sys.self user.child sys.child
## 1        rPar(10)          5   0.022    1.000    0.020   0.000           0         0
## 3        rPar(25)          5   0.034    1.545    0.036   0.000           0         0
## 2        rRNG(10)          5   0.039    1.773    0.040   0.000           0         0
## 5        rPar(50)          5   0.055    2.500    0.056   0.000           0         0
## 4        rRNG(25)          5   0.060    2.727    0.056   0.004           0         0
## 6        rRNG(50)          5   0.100    4.545    0.100   0.000           0         0
## 9  rPar(10, 0.05)          5   2.633  119.682    0.124   0.000           0         0
## 10 rRNG(10, 0.05)          5   2.698  122.636    0.192   0.000           0         0
## 7  rPar(50, 0.01)          5   2.979  135.409    0.436   0.004           0         0
## 8  rRNG(50, 0.01)          5   3.129  142.227    0.580   0.004           0         0
```

## 7 Known issues

- Nested and/or conditional foreach loops using the operator `%:%` are not currently not supported (see Section 5 for a work around).
- An error is thrown in `doRNG` 1.2.6, when the package `iterators` was not loaded, when used with `foreach` ¿= 1.4.
- There was a bug in versions prior to `1.4`, which caused `set.seed` and `.options.RNG` not to be equivalent when the current RNG was `"L'Ecuyer-CMRG"`. This behaviour can still be reproduced by setting:

  ```
  doRNGversion('1.3')
  ```

  To revert to the latest default behaviour:

  ```
  doRNGversion(NULL)
  ```

## 8 News and changes

```
*************************************************************************
Changes in version 1.7
*************************************************************************
CHANGES
    o Unit tests are now run through testthat
    o Minor fixes in man pages and README file
    o Now depends on rngtools >= 1.3

FIXES
    o Enabled running %dorng% loops within a package (incorporating
    the solution proposed by Elizabeth Byerly in PR#3)

*************************************************************************
Changes in version 1.6.2
*************************************************************************

FIXES
    o Non reproducible %dorng% loop when doRNG is registered over doSEQ
```

```
    (Issue #1 reported by Brenton Kenkel). Actually due to %dorng% not
    restoring the RNG (to state + 1) when doRNG is registered over doSEQ.
    o %dorng% was not working properly on loops of length one (Issue #2)

*************************************************************************
Changes in version 1.6
*************************************************************************

CHANGES
    o doRNG now depends on the package pkgmaker (>= 0.20)

FIXES
    o Check error due number of cores used. Now limited to 2 in examples,
    vignette and unit test.

*************************************************************************
Changes in version 1.5
*************************************************************************

CHANGES
    o doRNG now depends on the package pkgmaker (>= 0.9)
    o improved vignette
    o most of the general RNG utilities have been incorporated in a new
    package called rngtools.


*************************************************************************
Changes in version 1.4.1
*************************************************************************

CHANGES
    o when the current RNG was L'Ecuyer-CMRG, unseeded loops now use
    the current RNG stream as for the first stream in the RNG sequence
    and changes the current RNG to the next RNG stream of the last stream
    in the sequence.

BUG FIX
    o fix error "'iter' not found" due to changes in foreach package
    dependencies -- that was announced by Rich Calaway.
    o loops seeded with set.seed and .options.RNG were not reproducible
    when current RNG was L'Ecuyer-CMRG (reported by Zhang Peng)
    o separate unseeded loops were sharing most of their streams,
    when current RNG was L'Ecuyer-CMRG the RNG seed.
    o nested/conditional loops were crashing with a bad error.
    They are still not supported but the error message is nicer and a
    work around has been added to the vignette (reported by Chanhee Yi
    and Zhang Peng).

*************************************************************************
Changes in version 1.2.3
*************************************************************************

BUG FIX
    o fixed error when running a %dorng% loop on a fresh session, with no
    parallel backend registered.

CHANGES
    o improved vignette
    o added more unit tests
    o changed the name of the RNG attribute on result of %dorng% looops
    from 'RNG' to 'rng'. It now contains the whole sequence of RNG seeds,
    instead of only the first one.
    o RNGseq now accepts a list or a matrix describing the whole sequence
    of seeds. See vignette for more details.
    o %dorng% loops can be seeded with a complete sequence of seeds passed
    as a list, a matrix, or an object with attribute 'rng', e.g. the
    results of %dorng% loops. See vignette for more details.

*****************************************************************************
Changes in version 1.2.2
*****************************************************************************

BUG FIX
    o separate %dorng% loops were using the same seed.

NEW FEATURES
    o add unit tests
    o first seed is set as an attribute of the loop's result

CHANGES
    o function doRNGseed now returns the seed to use for the first
    iteration.
    o RNGseq now change the current RNG state if called with no seed
    specific.

DEFUNCT
    o removed function CMRGseed

*****************************************************************************
Changes in version 1.2
*****************************************************************************

BUG FIX
    o An error was thrown if using %dorng% loops before using any random
    generator. Thanks to Eric Lehmann for reporting this.

CHANGES
    o add vignette
    o use package doParallel in examples

*****************************************************************************
Changes in version 1.1
*****************************************************************************

CHANGES
    o use R core RNG "L'Ecuyer-CMRG" and the parallel package,
    instead of the implementation provided by the rstream package.
```

## Cleanup

```r
stopCluster(cl)
```

## Session information

```
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.4 LTS
```

```
Matrix products: default
BLAS: /usr/lib/openblas-base/libblas.so.3
LAPACK: /usr/lib/libopenblasp-r0.2.18.so

locale:
 [1] LC_CTYPE=en_ZA.UTF-8       LC_NUMERIC=C               LC_TIME=en_ZA.UTF-8
 [4] LC_COLLATE=C               LC_MONETARY=en_ZA.UTF-8    LC_MESSAGES=en_ZA.UTF-8
 [7] LC_PAPER=en_ZA.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_ZA.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] rbenchmark_1.0.0  doRNG_1.7.1       rngtools_1.3      doParallel_1.0.11
[5] iterators_1.0.9   foreach_1.4.4     knitr_1.20        pkgmaker_0.27.1
[9] registry_0.5

loaded via a namespace (and not attached):
 [1] codetools_0.2-15 digest_0.6.15    withr_2.1.2      assertthat_0.2.0 xtable_1.8-2
 [6] magrittr_1.5     evaluate_0.10.1  bibtex_0.4.2     highr_0.6        stringi_1.2.2
[11] tools_3.4.4      stringr_1.3.1    compiler_3.4.4
```