Using the **doRNG** package

doRNG package – Version 1.3

Renaud Gaujoux

May 22, 2012

Contents

Introduction

Research reproducibility is an issue of concern, e.g. in bioinformatics [4, 9, 5]. 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] [2] provides a very convenient way to perform parallel computations, with different parallel environments such as MPI or Redis, using a transparent loop-like syntax:

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

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

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

```r
cl <- makeCluster(2)
registerDoParallel(cl)

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

## [1] 1.514 2.775 3.388 4.832 5.991
```

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:

```r
# 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 L'Ecuyer [6]. 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] [3] 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 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 [10, 1], but should work with other backends such as provided by the *doRedis* package[5] [7].

[2] `http://cran.r-project.org/package=doRNG`
[3] `http://cran.r-project.org/package=doParallel`
[4] `http://cran.r-project.org/package=doMPI`
[5] `http://cran.r-project.org/package=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: methods


# 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"` [6], using the function `nextRNGStream` from the *parallel* package[6] [8], 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;

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
##
```

[6]http://cran.r-project.org/package=parallel

```
## [[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:

```
nextRNGstream(1:6)

## Error: could not find function "nextRNGstream"
```

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 classically within the call to RNGkind("L'Ecuyer-CMRG").

.options.RNG with single integer: the %dorng% operator implements an option that can be passed in the foreach statement, with the same arguments accepted by set.seed:

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

## [1] TRUE


## 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
```

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.

```r
# 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
cl <- makeCluster(3)
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    17752    17752    17752    17752    17752
##         17764    17773    17764    17773    17773
## res2    17786    17795    17805    17786    17805

# 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:

```
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 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(50), rRNG(50), rPar(100), rRNG(100), rPar(10,
    0.1), rRNG(10, 0.1), rPar(100, 0.01), rRNG(100, 0.01), replications = 50)
# order by increasing elapsed time
cmp[order(cmp$elapsed), ]
```

```
##               test replications elapsed relative user.self sys.self user.child sys.child
## 1         rPar(10)           50   0.563    1.000      0.57     0.00          0         0
## 2         rRNG(10)           50   0.865    1.536      0.85     0.02          0         0
## 3         rPar(50)           50   1.661    2.950      1.66     0.00          0         0
## 4         rRNG(50)           50   2.672    4.746      2.66     0.02          0         0
## 5        rPar(100)           50   3.099    5.504      3.09     0.00          0         0
## 6        rRNG(100)           50   4.994    8.870      4.95     0.04          0         0
## 7    rPar(10, 0.1)           50  50.743   90.130      0.69     0.00          0         0
## 8    rRNG(10, 0.1)           50  51.178   90.902      1.10     0.02          0         0
## 9  rPar(100, 0.01)           50  54.309   96.464      3.96     0.04          0         0
## 10 rRNG(100, 0.01)           50  56.405  100.187      6.01     0.06          0         0
```

6 Known issues

- Nested and/or conditional foreach loops using the operator `%:%` are not currently not supported.
- An error is thrown in `doRNG` < 1.3, when the package `iterators` was not loaded, when used with `foreach` >= 1.4.

Cleanup

```
stopCluster(cl)
```

Session information

```
R version 2.15.0 (2012-03-30)
Platform: x86_64-pc-linux-gnu (64-bit)

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

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

other attached packages:
[1] doRNG_1.3        doParallel_1.0.1 iterators_1.0.6  foreach_1.4.0    knitr_0.5

loaded via a namespace (and not attached):
 [1] codetools_0.2-8 compiler_2.15.0 digest_0.5.2    evaluate_0.4.2  formatR_0.4
 [6] highlight_0.3.1 parser_0.0-14   plyr_1.7.1      Rcpp_0.9.10     stringr_0.6
[11] tools_2.15.0
```

References

[1] Revolution Analytics. *doParallel: Foreach parallel adaptor for the parallel package.* R package version 1.0.1. 2012. URL: http://CRAN.R-project.org/package=doParallel.

[2] Revolution Analytics. *foreach: Foreach looping construct for R.* R package version 1.4.0. 2012. URL: http://CRAN.R-project.org/package=foreach.

[3] Renaud Gaujoux. *doRNG: Generic Reproducible Parallel Backend for foreach Loops.* R package version 1.3. 2012.

[4] Torsten Hothorn and Friedrich Leisch. "Case studies in reproducibility." In: *Briefings in bioinformatics* (Jan. 2011). ISSN: 1477-4054. DOI: 10.1093/bib/bbq084. URL: http://www.ncbi.nlm.nih.gov/pubmed/21278369.

[5] John P A Ioannidis et al. "The reproducibility of lists of differentially expressed genes in microarray studies". In: *Nature Genetics* 41.2 (2008), pp. 149–155. ISSN: 10614036. DOI: 10.1038/ng.295. URL: http://www.nature.com/doifinder/10.1038/ng.295.

[6] Pierre L'Ecuyer. "Good parameters and implementations for combined multiple recursive random number generators". In: *Operations Research* 47.1 (1999). URL: http://www.jstor.org/stable/10.2307/222902.

[7] B. W. Lewis. *doRedis: Foreach parallel adapter for the rredis package.* R package version 1.0.4. 2011. URL: http://CRAN.R-project.org/package=doRedis.

[8] R Development Core Team. *R: A Language and Environment for Statistical Computing.* ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2012. URL: http://www.R-project.org/.

[9] Victoria C Stodden. *The Digitization of Science: Reproducibility and Interdisciplinary Knowledge Transfer.* 2011. URL: http://aaas.confex.com/aaas/2011/webprogram/Session3166.html.

[10] Steve Weston. *doMPI: Foreach parallel adaptor for the Rmpi package.* R package version 0.1-5. 2010. URL: http://CRAN.R-project.org/package=doMPI.