Using the `doRNG` package

doRNG package – Version 1.8.6

Renaud Gaujoux

January 13, 2023

Contents

Introduction

Research reproducibility is an issue of concern, e.g. in bioinformatics [3, 9, 4]. 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] [7] 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] 1.464479 2.727211 3.601882 4.544061 5.801541
```

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 L'Ecuyer [5]. 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] [2] 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

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

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] [6].

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"` [5], 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:

[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

```
attr(res, 'rng')

## [[1]]
## [1]       10407   642048078    81368183 -2093158836   506506973  1421492218 -1906381517
##
## [[2]]
## [1]       10407  1340772676 -1389246211  -999053355  -953732024  1888105061  2010658538
##
## [[3]]
## [1]       10407 -1318496690  -948316584   683309249  -990823268 -1895972179  1275914972
##
## [[4]]
## [1]       10407   524763474  1715794407  1887051490 -1833874283   494155061 -1221391662
##
## [[5]]
## [1]       10407 -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:

```r
## 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`:

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

## [[1]]
## [1] 10407     1     2     3     4     5     6
##
## [[2]]
## [1]       10407  -447371532   542750874  -935969228  -269326340   701604884 -1748056907
##
## [[3]]
## [1]       10407   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]:

```r
# 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])
```

[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).

```
## [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]:

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

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

```r
# Marsaglia-Multicarry
RNGkind('Marsaglia')

## Warning in RNGkind("Marsaglia"):  RNGkind:  Marsaglia-Multicarry has poor statistical
properties

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.

```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
# 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  2767062  2767062  2767062  2767062  2767062
##       2767094  2767095  2767094  2767095  2767094
## res2  2767126  2767125  2767126  2767125  2767126

# 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]       10407   642048078    81368183 -2093158836   506506973  1421492218 -1906381517
##
## [[2]]
## [1]       10407  1340772676 -1389246211  -999053355  -953732024  1888105061  2010658538
##
## [[3]]
## [1]       10407 -1318496690  -948316584   683309249  -990823268 -1895972179  1275914972
##
## [[4]]
## [1]       10407   524763474  1715794407  1887051490 -1833874283   494155061 -1221391662
##
## [[5]]
## [1]       10407 -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
```

```r
    # 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:

```r
# 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:

```r
# 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   0.030    1.000     0.030   0.000           0          0
## 3        rPar(25)            5   0.055    1.833     0.056   0.000           0          0
## 2        rRNG(10)            5   0.069    2.300     0.070   0.000           0          0
## 5        rPar(50)            5   0.099    3.300     0.099   0.000           0          0
## 4        rRNG(25)            5   0.110    3.667     0.110   0.000           0          0
## 6        rRNG(50)            5   0.175    5.833     0.174   0.000           0          0
## 9  rPar(10, 0.05)            5   2.548   84.933     0.043   0.000           0          0
## 10 rRNG(10, 0.05)            5   2.568   85.600     0.062   0.000           0          0
## 7  rPar(50, 0.01)            5   2.687   89.567     0.149   0.008           0          0
## 8  rRNG(50, 0.01)            5   2.731   91.033     0.207   0.000           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

```
\begin{kframe}

{\ttfamily\noindent\color{warningcolor}{\#\# Warning in file(con, "{}r"{}): file("{}"{}) only supports open = "{}w+"{} and c
```

Cleanup

```
stopCluster(cl)
```

Session information

```
R version 4.2.1 (2022-06-23)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 22.10

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.1

Random number generation:
 RNG:     L'Ecuyer-CMRG
 Normal:  Inversion
 Sample:  Rejection

locale:
 [1] LC_CTYPE=en_IL       LC_NUMERIC=C         LC_TIME=en_IL        LC_COLLATE=C
 [5] LC_MONETARY=en_IL    LC_MESSAGES=en_IL    LC_PAPER=en_IL       LC_NAME=C
 [9] LC_ADDRESS=C         LC_TELEPHONE=C       LC_MEASUREMENT=en_IL LC_IDENTIFICATION=C

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

other attached packages:
[1] rbenchmark_1.0.0  doRNG_1.8.6       rngtools_1.5.2    doParallel_1.0.17
[5] iterators_1.0.14  foreach_1.5.2     knitr_1.41        pkgmaker_0.32.7
[9] registry_0.5-1

loaded via a namespace (and not attached):
 [1] codetools_0.2-18 withr_2.5.0      digest_0.6.31    assertthat_0.2.1 xtable_1.8-4
 [6] lifecycle_1.0.3  magrittr_2.0.3   evaluate_0.19    highr_0.10       rlang_1.0.6
[11] stringi_1.7.8    cli_3.5.0        vctrs_0.5.1      tools_4.2.1      stringr_1.5.0
[16] glue_1.6.2       xfun_0.36        compiler_4.2.1
```

References

[1] Microsoft Corporation and Steve Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.17. 2022. URL: https://CRAN.R-project.org/package=doParallel.

[2] Renaud Gaujoux. *doRNG: Generic Reproducible Parallel Backend for 'foreach' Loops*. R package version 1.8.6. URL: https://renozao.github.io/doRNG/.

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

[4] 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.

[5] Pierre L’Ecuyer. “Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators”. In: *Operations Research* 47.1 (1999), pp. 159–164. ISSN: 0030-364X. DOI: 10.1287/opre.47.1.159. URL: http://www.jstor.org/stable/10.2307/222902http://pubsonline.informs.org/doi/abs/10.1287/opre.47.1.159.

[6] B. W. Lewis. *doRedis: 'Foreach' Parallel Adapter Using the 'Redis' Database*. R package version 3.0.1. 2022. URL: https://CRAN.R-project.org/package=doRedis.

[7] Microsoft and Steve Weston. *foreach: Provides Foreach Looping Construct*. R package version 1.5.2. 2022. URL: https://CRAN.R-project.org/package=foreach.

[8] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2022. URL: https://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 0.2.2. 2017. URL: https://CRAN.R-project.org/package=doMPI.