Random Numbers in BiocParallel

Martin Morgan[1]

[1]Martin.Morgan@RoswellPark.org

Edited: 7 September, 2021; Compiled: October 11, 2022

Contents

1 Scope 1

2 Essentials 2

2.1 Use of `bplapply()` and `RNGseed=` 2

2.2 Use with `bpiterate()` 3

2.3 Use with `bptry()` 4

2.4 Relationship between `RNGseed=` and `set.seed()` 5

2.5 `bpstart()` and random number streams 6

2.6 Relationship between `bplapply()` and `lapply()` 6

3 Implementation notes 7

4 `sessionInfo()` 7

1 Scope

BiocParallel enables use of random number streams in a reproducible manner. This document applies to the following `*Param()`:

- `SerialParam()`: sequential evaluation in a single *R* process.
- `SnowParam()`: parallel evaluation in multiple independent *R* processes.
- `MulticoreParam())`: parallel evaluation in *R* sessions running in forked threads. Not available on Windows.

The `*Param()` can be used for evaluation with:

- `bplapply()`: `lapply()`-like application of a user-supplied function `FUN` to a vector or list of elements `X`.
- `bpiterate()`: apply a user-supplied function `FUN` to an unknown number of elements resulting from successive calls to a user-supplied function `ITER`.

The reproducible random number implementation also supports:

- `bptry()` and the `BPREDO=` argument, for re-evaluation of elements that fail (e.g., because of a bug in `FUN`).

2 Essentials

2.1 Use of `bplapply()` and `RNGseed=`

Attach *BiocParallel* and ensure that the version is greater than 1.27.5

```
library(BiocParallel)
stopifnot(
    packageVersion("BiocParallel") > "1.27.5"
)
```

For reproducible calculation, use the `RNGseed=` argument in any of the `*Param()` constructors.

```
result1 <- bplapply(1:3, runif, BPPARAM = SerialParam(RNGseed = 100))
result1

## [[1]]
## [1] 0.7393338
##
## [[2]]
## [1] 0.8216743 0.7451087
##
## [[3]]
## [1] 0.1962909 0.5226640 0.6857650
```

Repeating the calculation with the same value for `RNGseed=` results in the same result; a different random number seed results in different results.

```
result2 <- bplapply(1:3, runif, BPPARAM = SerialParam(RNGseed = 100))
stopifnot(
    identical(result1, result2)
)

result3 <- bplapply(1:3, runif, BPPARAM = SerialParam(RNGseed = 200))
result3

## [[1]]
## [1] 0.9757768
##
## [[2]]
## [1] 0.6525851 0.6416909
##
## [[3]]
## [1] 0.6710576 0.5895330 0.7686983

stopifnot(
    !identical(result1, result3)
)
```

Results are invariant across `*Param()`

```
result4 <- bplapply(1:3, runif, BPPARAM = SnowParam(RNGseed = 100))
stopifnot(
    identical(result1, result4)
)

if (!identical(.Platform$OS.type, "windows")) {
    result5 <- bplapply(1:3, runif, BPPARAM = MulticoreParam(RNGseed = 100))
    stopifnot(
        identical(result1, result5)
    )
}
```

Parallel backends can adjust the number of `workers` (processes performing the evaluation) and `tasks` (how elements of `X` are distributed between workers). Results are invariant to these parameters. This is illustrated with `SnowParam()`, but applies also to `MulticoreParam()`.

```
result6 <- bplapply(1:3, runif, BPPARAM = SnowParam(workers = 2, RNGseed = 100))
result7 <- bplapply(1:3, runif, BPPARAM = SnowParam(workers = 3, RNGseed = 100))
result8 <- bplapply(
    1:3, runif,
    BPPARAM = SnowParam(workers = 2, tasks = 3, RNGseed = 100)
)
stopifnot(
    identical(result1, result6),
    identical(result1, result7),
    identical(result1, result8)
)
```

Subsequent sections illustrate results with `SerialParam()`, but identical results are obtained with `SnowParam()` and `MulticoreParam()`.

2.2 Use with `bpiterate()`

`bpiterate()` allows parallel processing of a 'stream' of data as a series of tasks, with a task consisting of a portion of the overall data. It is useful when the data size is not known or easily partitioned into elements of a vector or list. A real use case might involve iterating through a BAM file, where a task represents successive records (perhaps 100,000 per task) in the file. Here we illustrate with a simple example – iterating through a vector `x = 1:3`

```
ITER_FUN_FACTORY <- function() {
    x <- 1:3
    i <- 0L
    function() {
        i <<- i + 1L
        if (i > length(x))
            return(NULL)
        x[[i]]
    }
}
```

`ITER_FUN_FACTORY()` is used to create a function that, on each invocation, returns the next task (here, an element of `x`; in a real example, perhaps 100000 records from a BAM file). When there are no more tasks, the function returns `NULL`

```
ITER <- ITER_FUN_FACTORY()
ITER()

## [1] 1

ITER()

## [1] 2

ITER()

## [1] 3

ITER()

## NULL
```

In our simple example, `bpiterate()` is performing the same computations as `bplapply()` so the results, including the random number streams used by each task in `bpiterate()`, are the same

```
result9 <- bpiterate(
    ITER_FUN_FACTORY(), runif,
    BPPARAM = SerialParam(RNGseed = 100)
)
stopifnot(
    identical(result1, result9)
)
```

2.3 Use with `bptry()`

`bptry()` in conjunction with the `BPREDO=` argument to `bplapply()` or `bpiterate()` allows for graceful recovery from errors. Here a buggy `FUN1()` produces an error for the second element. `bptry()` allows evaluation to continue for other elements of `X`, despite the error. This is shown in the result.

```
FUN1 <- function(i) {
    if (identical(i, 2L)) {
        ## error when evaluating the second element
        stop("i == 2")
    } else runif(i)
}
result10 <- bptry(bplapply(
    1:3, FUN1,
    BPPARAM = SerialParam(RNGseed = 100, stop.on.error = FALSE)
))
result10

## [[1]]
## [1] 0.7393338
##
## [[2]]
## <remote_error in FUN(...): i == 2>
## traceback() available as 'attr(x, "traceback")'
##
```

```
## [[3]]
## [1] 0.1962909 0.5226640 0.6857650
##
## attr(,"REDOENV")
## <environment: 0x5633cacb8728>
```

`FUN2()` illustrates the flexibility of `bptry()` by fixing the bug when `i == 2`, but also generating incorrect results if invoked for previously correct values. The identity of the result to the original computation shows that only the error task is re-computed, and that the random number stream used by the task is identical to the original stream.

```
FUN2 <- function(i) {
    if (identical(i, 2L)) {
        ## the random number stream should be in the same state as the
        ## first time through the loop, and rnorm(i) should return
        ## same result as FUN
        runif(i)
    } else {
        ## if this branch is used, then we are incorrectly updating
        ## already calculated elements -- '0' in the output would
        ## indicate this error
        0
    }
}
result11 <- bplapply(
    1:3, FUN2,
    BPREDO = result10,
    BPPARAM = SerialParam(RNGseed = 100, stop.on.error = FALSE)
)
stopifnot(
    identical(result1, result11)
)
```

2.4 Relationship between `RNGseed=` and `set.seed()`

The global random number stream (influenced by `set.seed()`) is ignored by *BiocParallel*, and *BiocParallel* does NOT increment the global stream.

```
set.seed(200)
value <- runif(1)

set.seed(200)
result12 <- bplapply(1:3, runif, BPPARAM = SerialParam(RNGseed = 100))
stopifnot(
    identical(result1, result12),
    identical(value, runif(1))
)
```

When `RNGseed=` is not used, an internal stream (not accessible to the user) is used and *BiocParallel* does NOT increment the global stream.

```
set.seed(100)
value <- runif(1)

set.seed(100)
result13 <- bplapply(1:3, runif, BPPARAM = SerialParam())
stopifnot(
    !identical(result1, result13),
    identical(value, runif(1))
)
```

2.5 `bpstart()` and random number streams

In all of the examples so far `*Param()` objects are passed to `bplapply()` or `bpiterate()` in the 'stopped' state. Internally, `bplapply()` and `bpiterate()` invoke `bpstart()` to establish the computational environment (e.g., starting workers for `SnowParam()`). `bpstart()` can be called explicitly, e.g., to allow workers to be used across calls to `bplapply()`.

The cluster random number stream is initiated with `bpstart()`. Thus

```
param <- bpstart(SerialParam(RNGseed = 100))
result16 <- bplapply(1:3, runif, BPPARAM = param)
bpstop(param)
stopifnot(
    identical(result1, result16)
)
```

This allows a second call to `bplapply` to represent a continuation of a random number computation – the second call to `bplapply()` results in different random number streams for each element of `X`.

```
param <- bpstart(SerialParam(RNGseed = 100))
result16 <- bplapply(1:3, runif, BPPARAM = param)
result17 <- bplapply(1:3, runif, BPPARAM = param)
bpstop(param)
stopifnot(
    identical(result1, result16),
    !identical(result1, result17)
)
```

2.6 Relationship between `bplapply()` and `lapply()`

The results from `bplapply()` are different from the results from `lapply()`, even with the same random number seed. This is because correctly implemented parallel random streams require use of a particular random number generator invoked in specific ways for each element of `X`, as outlined in the Implementation notes section.

```
set.seed(100)
result20 <- lapply(1:3, runif)
stopifnot(
    !identical(result1, result20)
)
```

3 Implementation notes

The implementation uses the L'Ecuyer-CMRG random number generator (see `?RNGkind` and `?parallel::clusterSetRNGStream` for additional details). This random number generates independent streams and substreams of random numbers. In *BiocParallel*, each call to `bpstart()` creates a new stream from the L'Ecuyer-CMRG generator. Each element in `bplapply()` or `bpiterate()` creates a new substream. Each application of `FUN` is therefore using the L'Ecuyer-CMRG random number generator, with a substream that is independent of the substreams of all other elements.

Within the user-supplied `FUN` of `bplapply()` or `bpiterate()`, it is a mistake to use `RNGkind()` to set a different random number generator, or to use `set.seed()`. This would in principle compromise the independence of the streams across elements.

4 `sessionInfo()`

```
## R version 4.2.1 (2022-06-23)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.5 LTS
##
## Matrix products: default
## BLAS:   /home/biocbuild/bbs-3.15-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.15-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB              LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4    stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
##  [2] GenomicFeatures_1.48.4
##  [3] AnnotationDbi_1.58.0
##  [4] RNAseqData.HNRNPC.bam.chr14_0.34.0
##  [5] GenomicAlignments_1.32.1
##  [6] VariantAnnotation_1.42.1
##  [7] Rsamtools_2.12.0
##  [8] Biostrings_2.64.1
##  [9] XVector_0.36.0
## [10] SummarizedExperiment_1.26.1
## [11] Biobase_2.56.0
## [12] GenomicRanges_1.48.0
## [13] GenomeInfoDb_1.32.4
## [14] IRanges_2.30.1
## [15] S4Vectors_0.34.0
## [16] MatrixGenerics_1.8.1
## [17] matrixStats_0.62.0
## [18] BiocGenerics_0.42.0
## [19] BiocParallel_1.30.4
##
## loaded via a namespace (and not attached):
##  [1] httr_1.4.4             bit64_4.0.5            assertthat_0.2.1
##  [4] BiocManager_1.30.18    highr_0.9              BiocFileCache_2.4.0
##  [7] base64url_1.4          blob_1.2.3             BSgenome_1.64.0
## [10] GenomeInfoDbData_1.2.8 yaml_2.3.5             progress_1.2.2
## [13] pillar_1.8.1           RSQLite_2.2.18         backports_1.4.1
## [16] lattice_0.20-45        glue_1.6.2             digest_0.6.29
## [19] checkmate_2.1.0        htmltools_0.5.3        Matrix_1.5-1
## [22] XML_3.99-0.11          pkgconfig_2.0.3        biomaRt_2.52.0
## [25] zlibbioc_1.42.0        snow_0.4-4             brew_1.0-8
## [28] tibble_3.1.8           KEGGREST_1.36.3        generics_0.1.3
## [31] ellipsis_0.3.2         cachem_1.0.6           withr_2.5.0
## [34] cli_3.4.1              magrittr_2.0.3         crayon_1.5.2
## [37] memoise_2.0.1          evaluate_0.17          fs_1.5.2
## [40] fansi_1.0.3            xml2_1.3.3             tools_4.2.1
## [43] data.table_1.14.2      prettyunits_1.1.1      hms_1.1.2
## [46] BiocStyle_2.24.0       BiocIO_1.6.0           lifecycle_1.0.3
## [49] stringr_1.4.1          DelayedArray_0.22.0    compiler_4.2.1
## [52] rlang_1.0.6            debugme_1.1.0          grid_4.2.1
## [55] RCurl_1.98-1.9         rjson_0.2.21           rappdirs_0.3.3
## [58] bitops_1.0-7           rmarkdown_2.17         restfulr_0.0.15
## [61] codetools_0.2-18       curl_4.3.3             DBI_1.1.3
## [64] R6_2.5.1               rtracklayer_1.56.1     dplyr_1.0.10
## [67] knitr_1.40             fastmap_1.1.0          bit_4.0.4
## [70] utf8_1.2.2             filelock_1.0.2         stringi_1.7.8
## [73] parallel_4.2.1         Rcpp_1.0.9             vctrs_0.4.2
## [76] png_0.1-7              tidyselect_1.2.0       dbplyr_2.2.1
## [79] batchtools_0.9.15      xfun_0.33
```