Package 'doFuture'

June 6, 2025

Version 1.1.1

Title Use Foreach to Parallelize via the Future Framework

Depends foreach (>= 1.5.0), future (>= 1.49.0)

Imports future.apply, globals, iterators, parallel, utils

Suggests doRNG (>= 1.8.2), markdown, R.rsp

VignetteBuilder R.rsp

Description The 'future' package provides a unifying parallelization framework for R that supports many parallel and distributed backends <doi:10.32614/RJ-2021-048>. The 'foreach' package provides a powerful API for iterating over an R expression in parallel. The 'doFuture' package brings the best of the two together. There are two alternative ways to use this package. The recommended approach is to use 'y <- foreach(...) %dofuture% { ... }', which does not require using 'registerDoFuture()' and has many advantages over '%dopar%'. The alternative is the traditional 'foreach' approach by registering the 'foreach' adapter 'registerDoFuture()' and so that 'y <- foreach(...) %dopar% { ... }' runs in parallelizes with the 'future' framework.

License LGPL (>= 2.1)

LazyLoad TRUE

URL https://doFuture.futureverse.org, https://github.com/futureverse/doFuture

BugReports https://github.com/futureverse/doFuture/issues

Language en-US

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-7579-5165>)

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2025-06-06 05:50:02 UTC

Contents

`doFuture` *doFuture: Foreach Parallel Adapter using Futures*

Description

The **doFuture** package provides mechanisms for using the **foreach** package together with the **future** package such that `foreach()` parallelizes via *any* future backend.

Usage

There are two alternative ways to use this package:

1. `y <- foreach(...) %dofuture% { ... }`
2. `y <- foreach(...) %dopar% { ... }` with `registerDoFuture()`

The *first alternative* (recommended), which uses `%dofuture%`, avoids having to use `registerDoFuture()`. The `%dofuture%` operator provides a more consistent behavior than `%dopar%`, e.g. there is a unique set of foreach arguments instead of one per possible adapter. Identification of globals, random number generation (RNG), and error handling is handled by the future ecosystem, just like with other map-reduce solutions such as **future.apply** and **furrr**. An example is:

```
library(doFuture)
plan(multisession)

y <- foreach(x = 1:4, y = 1:10) %dofuture% {
  z <- x + y
  slow_sqrt(z)
}
```

This alternative is the recommended way to let `foreach()` parallelize via the future framework if you start out from scratch.

See `%dofuture%` for more details and examples on this approach.

The *second alternative* is based on the traditional **foreach** approach where one registers a foreach adapter to be used by `%dopar%`. A popular adapter is `doParallel::registerDoParallel()`, which parallelizes on the local machine using the **parallel** package. This package provides `registerDoFuture()`, which parallelizes using the **future** package, meaning any future-compliant parallel backend can be used. An example is:

```
library(doFuture)
registerDoFuture()
plan(multisession)

y <- foreach(x = 1:4, y = 1:10) %dopar% {
  z <- x + y
  slow_sqrt(z)
}
```

This alternative is useful if you already have a lot of R code that uses `%dopar%` and you just want to switch to using the future framework for parallelization. Using `registerDoFuture()` is also useful when you wish to use the future framework with packages and functions that uses `foreach()` and `%dopar%` internally, e.g. **caret**, **plyr**, **NMF**, and **glmnet**. It can also be used to configure the Bioconductor **BiocParallel** package, and any package that rely on it, to parallelize via the future framework.

See `registerDoFuture()` for more details and examples on this approach.

Author(s)

Maintainer: Henrik Bengtsson <henrikb@braju.com> (ORCID) [copyright holder]

See Also

Useful links:

- https://doFuture.futureverse.org
- https://github.com/futureverse/doFuture
- Report bugs at https://github.com/futureverse/doFuture/issues

---

`registerDoFuture` *Use the Foreach* `%dopar%` *Adapter with Futures*

Description

The `registerDoFuture()` function makes the `%dopar%` operator of the **foreach** package to process foreach iterations via any of the future backends supported by the **future** package, which includes various parallel and distributed backends. In other words, *if a computational backend is supported via the Future API, it'll be automatically available for all functions and packages making using the* **foreach** *framework.* Neither the developer nor the end user has to change any code.

Recommendation*: If you have the option, use* `%dofuture%` *instead of* `%dopar%`*, for all the benefits explained in its help page.*

Usage

```
registerDoFuture(flavor = c("%dopar%", "%dofuture%"))
```

Arguments

`flavor`	Control how the adapter should behave. If `"%dopar%"`, it behaves as a classical foreach adapter. If `"%dofuture%"`, it behaves as if `%dofuture%` would have been used instead of `%dopar%`.

Value

`registerDoFuture()` returns, invisibly, the previously registered foreach `%dopar%` backend.

Parallel backends

To use futures with the **foreach** package and its `%dopar%` operator, use `doFuture::registerDoFuture()` to register **doFuture** to be used as a `%dopar%` adapter. After this, `%dopar%` will parallelize with whatever **future** backend is set by `future::plan()`.

The built-in **future** backends are always available, e.g. sequential (sequential processing), multicore (forked processes), multisession (background R sessions), and cluster (background R sessions on local and remote machines). For example, `plan(multisession)` will make `%dopar%` parallelize via R processes running in the background on the local machine, and `plan(cluster, workers = c("n1", "n2", "n2", "n3"))` will parallelize via R processes running on external machines.

Additional backends are provided by other future-compliant packages. For example, the **future.batchtools** package provides support for high-performance compute (HPC) cluster schedulers such as SGE, Slurm, and TORQUE / PBS. As an illustration, `plan(batchtools_slurm)` will parallelize by submitting the foreach iterations as tasks to the Slurm scheduler, which in turn will distribute the tasks to one or more compute nodes.

Global variables and packages

Unless running locally in the global environment (= at the R prompt), the **foreach** package requires you do specify what global variables and packages need to be available and attached in order for the "foreach" expression to be evaluated properly. It is not uncommon to get errors on one or missing variables when moving from running a `res <- foreach() %dopar% { ... }` statement on the local machine to, say, another machine on the same network. The solution to the problem is to explicitly export those variables by specifying them in the `.export` argument to `foreach::foreach()`, e.g. `foreach(..., .export = c("mu", "sigma"))`. Likewise, if the expression needs specific packages to be attached, they can be listed in argument `.packages` of `foreach()`.

When using `registerDoFuture()`, the above becomes less critical, because by default the Future API identifies all globals and all packages automatically (via static code inspection). This is done exactly the same way regardless of future backend. This automatic identification of globals and packages is illustrated by the below example, which does *not* have specify `.export = c("my_stat")`. This works because the future framework detects that function `my_stat()` is needed and makes sure it is exported. If you would use, say, `cl <- parallel::makeCluster(2)` and `doParallel::registerDoParallel(cl)`, you would get a run-time error on `Error in { : task 1 failed - \"could not find function "my_stat" ...`.

Having said this, note that, in order for your "foreach" code to work everywhere and with other types of foreach adapters as well, you may want to make sure that you always specify arguments `.export` and `.packages`.

Load balancing ("chunking")

Whether load balancing ("chunking") should take place or not can be controlled by specifying either argument `.options.future = list(scheduling = <ratio>)` or `.options.future = list(chunk.size = <count>)` to `foreach()`.

The value `chunk.size` specifies the average number of elements processed per future ("chunks"). If `+Inf`, then all elements are processed in a single future (one worker). If `NULL`, then argument `future.scheduling` is used.

The value `scheduling` specifies the average number of futures ("chunks") that each worker processes. If `0.0`, then a single future is used to process all iterations; none of the other workers are not used. If `1.0` or `TRUE`, then one future per worker is used. If `2.0`, then each worker will process two futures (if there are enough iterations). If `+Inf` or `FALSE`, then one future per iteration is used. The default value is `scheduling = 1.0`.

The name of `foreach()` argument `.options.future` follows the naming conventions of the **doMC**, **doSNOW**, and **doParallel** packages, *This argument should not be mistaken for the* R *options of the future package*.

For backward-compatibility reasons with existing foreach code, one may also use arguments `.options.multicore = list(p` and `.options.snow = list(preschedule = <logical>)` when using **doFuture**. `.options.multicore = list(preschedule = TRUE)` is equivalent to `.options.future = list(scheduling = 1.0)` and `.options.multicore = list(preschedule = FALSE)` is equivalent to `.options.future = list(scheduling = +Inf)`. and analogously for `.options.snow`. Argument `.options.future` takes precedence over argument `.option.multicore` which takes precedence over argument `.option.snow`, when it comes to chunking.

Random Number Generation (RNG)

The doFuture adapter registered by `registerDoFuture()` does *not* itself provide a framework for generating proper random numbers in parallel. This is a deliberate design choice based on how the foreach ecosystem is set up and to align it with other foreach adapters, e.g. **doParallel**. To generate statistically sound parallel RNG, it is recommended to use the **doRNG** package, where the `%dorng%` operator is used in place of `%dopar%`. For example,

```
y <- foreach(i = 1:3) %dorng% {
  rnorm(1)
}
```

This works because **doRNG** is designed to work with any type of foreach `%dopar%` adapter including the one provided by **doFuture**.

If you forget to use `%dorng%` instead of `%dopar%` when the foreach iteration generates random numbers, **doFuture** will detect the mistake and produce an informative warning.

For package developers

Please refrain from modifying the foreach backend inside your package or functions, i.e. do not call any `registerNnn()` in your code. Instead, leave the control on what backend to use to the end user. This idea is part of the core philosophy of the **foreach** framework.

However, if you think it necessary to register the **doFuture** backend in a function, please make sure to undo your changes when exiting the function. This can be achieve by:

```
with(registerDoFuture(), local = TRUE)
...
```

This is important, because the end-user might have already registered a foreach backend elsewhere for other purposes and will most likely not known that calling your function will break their setup. *Remember, your package and its functions might be used in a greater context where multiple packages and functions are involved and those might also rely on the foreach framework, so it is important to avoid stepping on others' toes.*

Reporting on progress

How to report on progress is a frequently asked question, especially in long-running tasks and parallel processing. The **foreach** framework does *not* have a built-in mechanism for progress reporting(*).

When using **doFuture**, and the Futureverse in general, for processing, the **progressr** package can be used to signal progress updates in a near-live fashion. There is special argument related to `foreach()` or **doFuture** to achieve this. Instead, one calls a a, so called, "progressor" function within each iteration. See the **progressr** package and its `vignette(package = "progressr")` for examples.

(*) The legacy **doSNOW** package uses a special `foreach()` argument `.options.doSNOW$progress` that can be used to make a progress update each time results from a parallel workers is returned. This approach is limited by how chunking works, requires the developer to set that argument, and the code becomes incompatible with foreach adaptors registered by other **doNnn** packages.

Examples

```
library(iterators)  # iter()
registerDoFuture()  # (a) tell %dopar% to use the future framework
plan(multisession)  # (b) parallelize futures on the local machine


## Example 1
A <- matrix(rnorm(100^2), nrow = 100)
B <- t(A)

y1 <- apply(B, MARGIN = 2L, FUN = function(b) {
  A %*% b
})

y2 <- foreach(b = iter(B, by = "col"), .combine = cbind) %dopar% {
  A %*% b
}
stopifnot(all.equal(y2, y1))



## Example 2 - Chunking (4 elements per future [= worker])
y3 <- foreach(b = iter(B, by = "col"), .combine = cbind,
              .options.future = list(chunk.size = 10)) %dopar% {
  A %*% b
```

```
}
stopifnot(all.equal(y3, y1))


## Example 3 - Simulation with parallel RNG
library(doRNG)

my_stat <- function(x) {
  median(x)
}

my_experiment <- function(n, mu = 0.0, sigma = 1.0) {
  ## Important: use %dorng% whenever random numbers
  ##            are involved in parallel evaluation
  foreach(i = 1:n) %dorng% {
    x <- rnorm(i, mean = mu, sd = sigma)
    list(mu = mean(x), sigma = sd(x), own = my_stat(x))
  }
}

## Reproducible results when using the same RNG seed
set.seed(0xBEEF)
y1 <- my_experiment(n = 3)

set.seed(0xBEEF)
y2 <- my_experiment(n = 3)

stopifnot(identical(y2, y1))

## But only then
y3 <- my_experiment(n = 3)
str(y3)
stopifnot(!identical(y3, y1))
```

`with.DoPar` *Evaluate an Expression using a Temporarily Registered Foreach* `%dopar%` *Adapter*

Description

Evaluate an Expression using a Temporarily Registered Foreach `%dopar%` Adapter

Usage

```
## S3 method for class 'DoPar'
with(data, expr, ..., local = FALSE, envir = parent.frame())
```

Arguments

data	The foreach '%dopar% adapter to use temporarily.
expr	The R expression to be evaluated.
local	If TRUE, then the future plan specified by data is applied temporarily in the calling frame. Argument expr must not be specified if local = TRUE.
envir	The environment where the adapter should be set and the expression evaluated.
...	Not used.

Value

Invisibly, the value of expr.

Examples

```
with(registerDoFuture(), {
  y <- foreach(x = 1:3) %dopar% { x^2 }
})

a_fcn_in_a_pkg <- function(xs) {
  foreach(x = xs) %dopar% { x^2 }
}

with(registerDoFuture(flavor = "%dofuture%"), {
  y <- a_fcn_in_a_pkg(1:3)
})


my_fcn <- function(xs) {
  ## Use registerDoFuture() for this function only and then
  ## revert back to the previously set foreach adapter
  with(registerDoFuture(), local = TRUE)

  foreach(x = xs) %dopar% { x^2 }
}
```

---

withDoRNG *Evaluates a foreach %dopar% expression with the doRNG adapter*

Description

Evaluates a foreach %dopar% expression with the doRNG adapter

Usage

```
withDoRNG(expr, substitute = TRUE, envir = parent.frame())
```

Arguments

`expr`	An R expression.
`substitute`	(logical) If TRUE, `expr` is substituted, otherwise not.
`envir`	The environment where to evaluate `expr`.

Details

This function is useful when there is a foreach `%dopar%` expression that uses the random-number generator (RNG). Such code should ideally use `%doRNG%` of the **doRNG** package instead of `%dopar%`. Alternatively, and second best, is if the code would temporarily register the **doRNG** foreach adapter. If neither is done, then there is a risk that the random numbers are not statistically sound, e.g. they might be correlated. For what it is worth, the **doFuture** adapter, which is set by `registerDoFuture()`, detects when **doRNG** is forgotten, and produced an informative warning reminding us to use **doRNG**.

If you do not have control over the foreach code, you can use `withDoRNG()` to evaluate the foreach statement with `doRNG::registerDoRNG()` temporarily set.

Value

The value of `expr`.

Examples

Consider a function:

```
my_fcn <- function(n) {
  y <- foreach(i = seq_len(n)) %dopar% {
    stats::runif(n = 1L)
  }
  mean(unlist(y))
}
```

This function generates random numbers, but without involving **doRNG**, which risks generating poor randomness. If we call it as-is, with the **doFuture** adapter, we will get a warning about the problem:

```
> my_fcn(10)
[1] 0.5846141
Warning message:
UNRELIABLE VALUE: One of the foreach() iterations ('doFuture-1')
unexpectedly generated random numbers without declaring so. There is a
risk that those random numbers are not statistically sound and the overall
results might be invalid. To fix this, use '%dorng%' from the 'doRNG'
package instead of '%dopar%'. This ensures that proper, parallel-safe
random numbers are produced. To disable this check, set option
'doFuture.rng.onMisuse' to "ignore".
>
```

To fix this, we use `withDoRNG()` as:

```
> withDoRNG(my_fcn(10))
[1] 0.535326
```

`%dofuture%` *Loop over a Foreach Expression using Futures*

Description

Loop over a Foreach Expression using Futures

Usage

```
foreach %dofuture% expr
```

Arguments

`foreach`	A `foreach` object created by `foreach::foreach()` and `foreach::times()`.
`expr`	An R expression.

Details

This is a replacement for `%dopar%` of the **foreach** package that leverages the **future** framework.

When using `%dofuture%`:

- there is no need to use `registerDoFuture()`
- there is no need to use `%dorng%` of the **doRNG** package (but you need to specify `.options.future = list(seed = TRUE)` whenever using random numbers in the `expr` expression)
- global variables and packages are identified automatically by the **future** framework
- errors are relayed as-is with the default `.errorhandling = "stop"`, whereas with `%dopar%` they are captured and modified.

Value

The value of the foreach call.

Global variables and packages

When using `%dofuture%`, the future framework identifies globals and packages automatically (via static code inspection). However, there are cases where it fails to find some of the globals or packages. When this happens, one can specify the `future::future()` arguments `globals` and `packages` via foreach argument `.options.future`. For example, if you specify argument `.options.future = list(globals = structure(TRUE, ignore = "b", add = "a"))` then globals are automatically identified (`TRUE`), but it ignores `b` and always adds `a`.

An alternative to specifying the `globals` and the `packages` options via `.options.future`, is to use the `%globals%` and the `%packages%` operators. See the examples for an illustration.

For further details and instructions, see `future::future()`.

Random Number Generation (RNG)

The `%dofuture%` uses the future ecosystem to generate proper random numbers in parallel in the same way they are generated in, for instance, **future.apply**. For this to work, you need to specify `.options.future = list(seed = TRUE)`. For example,

```
y <- foreach(i = 1:3, .options.future = list(seed = TRUE)) %dofuture% {
  rnorm(1)
}
```

Unless `seed` is `FALSE` or `NULL`, this guarantees that the exact same sequence of random numbers are generated *given the same initial seed / RNG state* - this regardless of type of future backend, number of workers, and scheduling ("chunking") strategy.

RNG reproducibility is achieved by pregenerating the random seeds for all iterations by parallel RNG streams. In each iteration, these seeds are set before evaluating the foreach expression. *Note, for large number of iterations this may introduce a large overhead.*

If `seed = TRUE`, then `.Random.seed` is used if it holds a parallel RNG seed, otherwise one is created randomly.

If `seed = FALSE`, it is expected that none of the foreach iterations use random number generation. If they do, then an informative warning or error is produces depending on settings. See future::future for more details. Using `seed = NULL`, is like `seed = FALSE` but without the check whether random numbers were generated or not.

As input, `seed` may also take a fixed initial seed (integer), either as a full parallel RNG seed (vector of 1+6 integers), or as a seed generating such a full seed. This seed will be used to generated one parallel RNG stream for each iteration.

An alternative to specifying the `seed` option via `.options.future`, is to use the `%seed%` operator. See the examples for an illustration.

For further details and instructions, see `future.apply::future_lapply()`.

Load balancing ("chunking")

Whether load balancing ("chunking") should take place or not can be controlled by specifying either argument `.options.future = list(scheduling = <ratio>)` or `.options.future = list(chunk.size = <count>)` to `foreach()`.

The value `chunk.size` specifies the average number of elements processed per future ("chunks"). If `+Inf`, then all elements are processed in a single future (one worker). If `NULL`, then argument `future.scheduling` is used.

The value `scheduling` specifies the average number of futures ("chunks") that each worker processes. If `0.0`, then a single future is used to process all iterations; none of the other workers are not used. If `1.0` or `TRUE`, then one future per worker is used. If `2.0`, then each worker will process two futures (if there are enough iterations). If `+Inf` or `FALSE`, then one future per iteration is used. The default value is `scheduling = 1.0`.

For further details and instructions, see `future.apply::future_lapply()`.

Control processing order of iterations

Attribute `ordering` of `chunk.size` or `scheduling` can be used to control the ordering the elements are iterated over, which only affects the processing order and *not* the order values are returned. This attribute can take the following values:

- index vector - an numeric vector of length `nX`.
- function - an function taking one argument which is called as `ordering(nX)` and which must return an index vector of length `nX`, e.g. `function(n) rev(seq_len(n))` for reverse ordering.
- `"random"` - this will randomize the ordering via random index vector `sample.int(nX)`.

where `nX` is the number of foreach iterations to be done.

For example, `.options.future = list(scheduling = structure(2.0, ordering = "random"))`.

Note, when elements are processed out of order, then captured standard output and conditions are also relayed in that order, that is, out of order.

For further details and instructions, see `future.apply::future_lapply()`.

Reporting on progress

How to report on progress is a frequently asked question, especially in long-running tasks and parallel processing. The **foreach** framework does *not* have a built-in mechanism for progress reporting(*).

When using **doFuture**, and the Futureverse in general, for processing, the **progressr** package can be used to signal progress updates in a near-live fashion. There is special argument related to `foreach()` or **doFuture** to achieve this. Instead, one calls a a, so called, "progressor" function within each iteration. See the **progressr** package and its `vignette(package = "progressr")` for examples.

(*) The legacy **doSNOW** package uses a special `foreach()` argument `.options.doSNOW$progress` that can be used to make a progress update each time results from a parallel workers is returned. This approach is limited by how chunking works, requires the developer to set that argument, and the code becomes incompatible with foreach adaptors registered by other **doNnn** packages.

Examples

```
plan(multisession)  # parallelize futures on the local machine

y <- foreach(x = 1:10, .combine = rbind) %dofuture% {
  y <- sqrt(x)
  data.frame(x = x, y = y, pid = Sys.getpid())
}
print(y)


## Random number generation
y <- foreach(i = 1:3, .combine = rbind, .options.future = list(seed = TRUE)) %dofuture% {
  data.frame(i = i, random = runif(n = 1L))
}
print(y)
```

```
## Random number generation (alternative specification)
y <- foreach(i = 1:3, .combine = rbind) %dofuture% {
  data.frame(i = i, random = runif(n = 1L))
} %seed% TRUE
print(y)

## Random number generation with the foreach() %:% nested operator
y <- foreach(i = 1:3, .combine = rbind) %:%
      foreach(j = 3:5, .combine = rbind, .options.future = list(seed = TRUE)) %dofuture% {
  data.frame(i = i, j = j, random = runif(n = 1L))
}
print(y)

## Random number generation with the nested foreach() calls
y <- foreach(i = 1:3, .combine = rbind, .options.future = list(seed = TRUE)) %dofuture% {
  foreach(j = 3:5, .combine = rbind, .options.future = list(seed = TRUE)) %dofuture% {
    data.frame(i = i, j = j, random = runif(n = 1L))
  }
}
print(y)
```

Index