Package: futurize (via r-universe)

November 23, 2026

Version 1.2.0

Title Parallelize Common Functions via One Magic Function

Description The futurize() function <doi:10.48550/arXiv.2601.17578> turns sequential map-reduce functions such as base::lapply(), purrr::map(), 'foreach::foreach() %do% { ... }' into concurrent alternatives, providing you with a simple, straightforward path to scalable parallel computing via the 'future' ecosystem <doi:10.32614/RJ-2021-048>. By combining this transpiler function with R's native pipe operator, you have a convenient way for speeding up iterative computations with minimal refactoring, e.g. 'lapply(xs, fcn) |> futurize()', 'purrr::map(xs, fcn) |> futurize()', and 'foreach::foreach(x = xs) %do% { fcn(x) } |> futurize()'. Other map-reduce packages that can be ``futurized'' are 'BiocParallel', 'plyr', 'crossmap', 'pbapply' packages. There is also support for a growing set of domain-specific packages on CRAN (e.g. 'boot', 'caret', 'DiceKriging', 'ez', 'fgsea', 'fwb', 'gamlss', 'glmmTMB', 'glmnet', 'kernelshap', 'lme4', 'metafor', 'mgcv', 'mlr3', 'mlr3tuning', 'modelsummary', 'parameters', 'partykit', 'pls', 'pvclust', 'riskRegression', 'rugarch', 'sandwich', 'seriation', 'shapr', 'Sim.DiffProc', 'SimDesign', 'stars', 'strucchange', 'SuperLearner', 'terra', 'tm', 'TSP', and 'vegan') and on Bioconductor (e.g. 'bumphunter', 'DESeq2', 'GenomicAlignments', 'GSVA', 'Rsamtools', 'scater', 'scuttle', 'SingleCellExperiment', and 'sva').

License Apache License (>= 2)

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

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

Depends R (>= 4.1.0), future (>= 1.80.0)

Imports utils

Suggests methods, future.apply (>= 1.21.0), foreach, doFuture (>= 1.4.0), purrr, furrr, crossmap, plyr, pbapply, BiocParallel,

boot, survival, bumphunter, caret, randomForest, DESeq2, DiceKriging, ez, fgsea, GenomicAlignments, Rsamtools, fwb, gamlss, glmmTMB, glmnet, GSVA, kernelshap, lme4, metafor, mgcv, mlr3, rpart, mlr3tuning, modelsummary, parameters (>= 0.29.1), partykit, pls, pvclust, riskRegression, rugarch, sandwich, scater, scuttle, seriation, shapr, Sim.DiffProc, SimDesign, SingleCellExperiment, stars, strucchange, SuperLearner, sva, terra (>= 1.9-25), tm, vegan, tools, commonmark, base64enc

VignetteBuilder futurize

Language en-US

Encoding UTF-8

Config/roxygen2/version 8.1.1

NeedsCompilation no

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

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-11-23 03:48:39 UTC

RemoteUrl https://github.com/cran/futurize

RemoteRef HEAD

RemoteSha e9a41c3b9482690c6735652dc0572750df57af57

Contents

futurize 2
futurize_options 7
futurize_supported_packages 9
zzz-futurize.options 10

Index **11**

`futurize` *Turn common R function calls into concurrent calls for parallel evaluation*

Description

Usage

```
futurize(
  expr,
  substitute = TRUE,
  options = futurize_options(...),
  ...,
  when = TRUE,
  eval = TRUE,
  envir = parent.frame()
)
```

Arguments

`expr`	An R expression, typically a function call to futurize. If FALSE, then futurization is disabled, and if TRUE, it is re-enabled.
`substitute`	If TRUE, argument `expr` is `substitute()`:d, otherwise not.
`options, ...`	Named options, passed to `futurize_options()`, controlling how futures are resolved.
`when`	If TRUE (default), the expression is futurized, otherwise not.
`eval`	If TRUE (default), the futurized expression is evaluated, otherwise it is returned.
`envir`	The environment from where global objects should be identified.

Value

Returns the value of the evaluated expression `expr`.

If `expr` is TRUE or FALSE, then a logical is returned indicating whether futurization was previously enabled or disabled.

Expression unwrapping

The transpilation mechanism includes logic to "unwrap" expressions enclosed in constructs such as `!`, `{ }`, `( )`, `local()`, `I()`, `identity()`, `invisible()`, `suppressMessages()`, `suppressWarnings()`, `suppressPackageStartupMessages()`, `withCallingHandlers()`, and `with()`. The transpiler descends through wrapping constructs until it finds a transpilable expression, avoiding the need to place `futurize()` inside such constructs. This allows for patterns like:

```
y <- {
  lapply(xs, fcn)
} |> suppressMessages() |> futurize()
```

avoiding having to write:

```
y <- {
  lapply(xs, fcn) |> futurize()
} |> suppressMessages()
```

Generic functions

Some functions can be futurized only via their S3 or S4 methods, e.g. `influence()` of the **stats** package dispatches to the `influence.merMod()` method of the **lme4** package. To identify which method is called, `futurize()` has to evaluate the first argument of the generic function. If the first argument is a variable, a formula, or a `list()` or `c()` of variables, it is evaluated directly. Otherwise, `futurize()` makes sure it is evaluated only once, by first assigning it to a temporary variable. For example,

```
inf <- influence(lmer(...)) |> futurize()
```

is evaluated as:

```
inf <- local({
  ...futurize.x <- lmer(...)
  influence(...futurize.x) |> futurize()
})
```

Because of this, methods that use `substitute()` or `match.call()` on the first argument, e.g. for labeling, see `...futurize.x`. To avoid this, assign the first argument to a variable yourself, e.g.

```
fit <- lmer(Reaction ~ Days + (Days | Subject), data = sleepstudy)
inf <- influence(fit) |> futurize()
```

Conditional futurization

It is possible to control whether futurization should take place at run-time. For example,

```
y <- lapply(xs, fun) |> futurize(when = { length(xs) >= 10 })
```

will be futurized, unless `length(xs)` is less than ten, in which case it is evaluated as:

```
y <- lapply(xs, fun)
```

Disable and re-enable all futurization

It is possible to globally disable the effect of all `futurize()` calls by calling `futurize(FALSE)`. The effect is as if `futurize()` was never applied. For example,

```
futurize(FALSE)
y <- lapply(xs, fun) |> futurize()
```

evaluates as:

```
y <- lapply(xs, fun)
```

To re-enable futurization, call `futurize(TRUE)`. Please note that it is only the end-user that may control whether futurization should be disabled and enabled. A package must *never* disable or enable futurization.

Required packages

Futurization of a function call is in some cases done via a package that implements the corresponding future-based alternative. For example, `lapply(xs, fcn) |> futurize()` is futurized via `future.apply::future_lapply()`, and `purrr::map(xs, fcn) |> futurize()` via `furrr::future_map()`. These packages must be installed for `futurize()` to work.

If you use `futurize()` in a package, the package must (1) declare these required packages as dependencies too, e.g.

```
Imports:
    futurize,
    future.apply
```

in the package `DESCRIPTION` file. It must also (2) import one of its functions in the package `NAMESPACE` file, e.g. `importFrom(future.apply, future_lapply)`.

Future labels

Futures created by `futurize()` are labeled automatically, so that each future identifies the package and function that created it, e.g. in error messages, event logs, and tools that inspect futures. The default labels follow the convention:

- `fz:<package>::<function>-%d` if the call is evaluated by several futures, e.g. one per chunk, where `%d` is replaced by the index of the future
- `fz:<package>::<function>` if the call is evaluated by a single future

For example, `lapply(xs, fcn) |> futurize()` labels its futures `fz:base::lapply-1`, `fz:base::lapply-2`, and so on, and `boot::boot()` is labeled `fz:boot::boot`. The `fz:` prefix identifies futures that were created by `futurize()`.

To use a different label, specify argument `label`, which takes precedence over the default label. For example,

```
y <- lapply(xs, fcn) |> futurize(label = "my-analysis-%d")
```

labels the futures `my-analysis-1`, `my-analysis-2`, and so on.

See Also

To see which CRAN and Bioconductor packages are supported, use `futurize_supported_packages()`. To see which functions a specific package supports, use `futurize_supported_functions()`.

Examples

```
xs <- list(1, 1:2, 1:2, 1:5)

# ----------------------------------------
# Base R apply functions
# ----------------------------------------
if (requireNamespace("future.apply", quietly = TRUE)) {
```

```
# Sequential lapply()
y <- lapply(X = xs, FUN = function(x) {
  sum(x)
})

# Parallel version
y <- lapply(X = xs, FUN = function(x) {
  sum(x)
}) |> futurize()
str(y)

} ## if (require ...)


# ----------------------------------------
# purrr map-reduce functions with pipes
# ----------------------------------------
if (require("purrr") && requireNamespace("furrr", quietly = TRUE)) {

# Sequential map()
y <- xs |> map(sum)

# Parallel version
y <- xs |> map(sum) |> futurize()
str(y)

} ## if (require ...)


# ----------------------------------------
# foreach map-reduce functions
# ----------------------------------------
if (require("foreach") && requireNamespace("doFuture", quietly = TRUE)) {

# Sequential foreach()
y <- foreach(x = xs) %do% {
  sum(x)
}

# Parallel version
y <- foreach(x = xs) %do% {
  sum(x)
} |> futurize()
str(y)


# Sequential times()
y <- times(3) %do% rnorm(1)
str(y)

# Parallel version
y <- times(3) %do% rnorm(1) |> futurize()
str(y)
```

```
} ## if (require ...)


# ----------------------------------------
# plyr map-reduce functions
# ----------------------------------------
if (require("plyr") && requireNamespace("doFuture", quietly = TRUE)) {

# Sequential llply()
y <- llply(xs, sum)

# Parallel version
y <- llply(xs, sum) |> futurize()
str(y)

} ## if (require ...)
```

`futurize_options` *Options controlling resources, scheduling and evaluation of futures*

Description

Some futures require specific **resources** and capabilities (e.g., RNG, packages, and globals) in order to be evaluated. These can be declared via this function. In addition, this function can control how futures are **evaluated**, including what is collected from futures (e.g., standard output and conditions). It can also control how future are **scheduled**, including how a set of futures are partitioned (e.g. chunking).

Usage

```
futurize_options(
  rng = FALSE,
  seed = FALSE,
  globals = TRUE,
  packages = NULL,
  stdout = TRUE,
  conditions = "condition",
  scheduling = 1,
  chunk_size = NULL,
  ...
)
```

Arguments

`rng`	(resources) If TRUE, parallel-safe random number generation (RNG) is set up such that each element is processed with its own, statistically sound RNG stream. This is needed whenever the code uses random numbers, e.g. via `rnorm()` or `sample()`. The results are reproducible given a fixed `set.seed()`, regardless

of how and where the elements are processed. Currently, the L'Ecuyer-CMRG RNG is used for this. If FALSE (default), then no RNG streams are set up, and you will be warned if random numbers are still generated. If NA, then no RNG streams are set up, and there is no check for random numbers being generated. This can be used to silence false-positive RNG warnings, e.g. when the code uses random numbers in a way that does not affect the results. If in doubt, use `rng = TRUE`, which is always statistically sound. Currently, only `TRUE`, `FALSE`, and `NA` are supported; other types of values are reserved for future extensions. If `rng` is specified, then `seed` must not be specified.

`seed` (resources) Superseded by `rng`, but kept for backward compatibility. `seed = TRUE`, `seed = FALSE`, and `seed = NULL` are equivalent to `rng = TRUE`, `rng = FALSE`, and `rng = NA`. For other values, see the `seed` argument of `future::future()`.

`globals` (resource; optional) a logical, a character vector, or a named list to control how globals are handled. For details, see section 'Globals used by future expressions' in the help for `future()`.

`packages` (resource; optional) a character vector specifying packages to be attached in the R environment evaluating the future, *in addition to packages required by global variables* specified or identified via argument `globals`.

`stdout` (evaluation) If TRUE (default), then the standard output is captured, and re-outputted when `value()` is called. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). Using `stdout = structure(TRUE, drop = TRUE)` causes the captured standard output to be dropped from the future object as soon as it has been relayed. This can help decrease the overall memory consumed by captured output across futures. Using `stdout = NA` fully avoids intercepting the standard output; behavior of such unhandled standard output depends on the future backend.

`conditions` (evaluation) A character string of condition classes to be captured and relayed. The default is to relay all conditions, including messages and warnings. To drop all conditions, use `conditions = character(0)`. Errors are always relayed. Attribute `exclude` can be used to ignore specific classes, e.g. `conditions = structure("condition", exclude = "message")` will capture all `condition` classes except those that inherit from the `message` class. Using `conditions = structure(..., drop = TRUE)` causes any captured conditions to be dropped from the future object as soon as they have been relayed, e.g. by `value(f)`. This can help decrease the overall memory consumed by captured conditions across futures. Using `conditions = NULL` (not recommended) avoids intercepting conditions, except from errors; behavior of such unhandled conditions depends on the future backend and the environment from which R runs.

`scheduling` (scheduling) Average number of futures ("chunks") per worker. If `0.0`, then a single future is used to process all elements. 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 elements). If `Inf` or `FALSE`, then one future per element is used. Only used if `chunk_size` is `NULL`.

`chunk_size` (scheduling) The average number of elements per future ("chunk"). If `Inf`, then all elements are processed in a single future. If `NULL`, then argument `scheduling` is used.

`...` Additional named options.

Value

A named list of future options. Attribute `specified` is a character vector of future options that were explicitly specified.

Examples

```
# Default futurize options
str(futurize_options())

# Declare that parallel random number generation is needed
str(futurize_options(rng = TRUE))
```

---

futurize_supported_packages

List packages and functions supporting futurization

Description

List packages and functions supporting futurization

Usage

```
futurize_supported_packages()

futurize_supported_functions(package)
```

Arguments

package	A package name.

Value

A character vector of package or function names. `futurize_supported_functions()` produces an error if packages required by the futurize transpiler are not installed.

Examples

```
pkgs <- futurize_supported_packages()
pkgs

if (requireNamespace("future.apply")) {
  fcns <- futurize_supported_functions("base")
  print(fcns)
}

if (requireNamespace("doFuture")) {
  fcns <- futurize_supported_functions("foreach")
  print(fcns)
```

```
}
```

`zzz-futurize.options` *Options used by futurize*

Description

Below are the R options and environment variables that are used by the **futurize** package and packages enhancing it.

WARNING: Note that the names and the default values of these options may change in future versions of the package. Please use with care until further notice.

Packages must not change futurize options

Just like for other R options, as a package developer you must *not* change any of the below `futurize.*` options. Only the end-user should set these. If you find yourself having to tweak one of the options, make sure to undo your changes immediately afterward.

Options for debugging

'`futurize.debug`': (logical) If `TRUE`, extensive debug messages are generated. (Default: `FALSE`)

Options for controlling futurization

'`futurize.enable`': (logical) If `TRUE` (default), `futurize()` transpilation will be applied, otherwise not.

Environment variables that set R options

All of the above R '`futurize.*`' options can be set by corresponding environment variable `R_FUTURIZE_*` *when the* **futurize** *package is loaded*. This means that those environment variables must be set before the **futurize** package is loaded in order to have an effect. For example, if `R_FUTURIZE_DEBUG=true`, then option '`futurize.debug`' is set to `TRUE` (logical).

See Also

To set R options or environment variables when R starts (even before the **futurize** package is loaded), see the Startup help page. The **startup** package provides a friendly mechanism for configuring R's startup process.

Index

environment, *3*

future(), *8*
future::future(), *8*
futurize, 2
futurize.debug *(*zzz-futurize.options*)*, 10
futurize.enable *(*zzz-futurize.options*)*, 10
futurize.options *(*zzz-futurize.options*)*, 10
futurize_options, 7
futurize_options(), *3*
futurize_supported_functions *(*futurize_supported_packages*)*, 9
futurize_supported_functions(), *5*
futurize_supported_packages, 9
futurize_supported_packages(), *5*
fz *(*futurize*)*, 2

R_FUTURIZE_DEBUG *(*zzz-futurize.options*)*, 10
R_FUTURIZE_ENABLE *(*zzz-futurize.options*)*, 10

Startup, *10*
substitute, *3*

zzz-futurize.options, 10