Package 'future.apply'

December 9, 2025

Version 1.20.1

Title Apply Function to Elements in Parallel using Futures

Depends R (>= 3.2.0), future (>= 1.49.0)

Imports globals, parallel, utils

Suggests datasets, stats, tools, listenv, R.rsp, markdown

VignetteBuilder R.rsp

Description Implementations of apply(), by(), eapply(), lapply(), Map(), .mapply(), mapply(), replicate(), sapply(), tapply(), and vapply() that can be resolved using any future-supported backend, e.g. parallel on the local machine or distributed on a compute cluster. These future_*apply() functions come with the same pros and cons as the corresponding base-R *apply() functions but with the additional feature of being able to be processed via the future framework <doi:10.32614/RJ-2021-048>.

License GPL (>= 2)

LazyLoad TRUE

URL https://future.apply.futureverse.org,
https://github.com/futureverse/future.apply

BugReports https://github.com/futureverse/future.apply/issues

Language en-US

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

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

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2025-12-09 07:50:07 UTC

Contents

`future.apply` *future.apply: Apply Function to Elements in Parallel using Futures*

Description

The **future.apply** packages provides parallel implementations of common "apply" functions provided by base R. The parallel processing is performed via the **future** ecosystem, which provides a large number of parallel backends, e.g. on the local machine, a remote cluster, and a high-performance compute cluster.

Details

Currently implemented functions are:

- `future_apply()`: a parallel version of apply()
- `future_by()`: a parallel version of by()
- `future_eapply()`: a parallel version of eapply()
- `future_lapply()`: a parallel version of lapply()
- `future_mapply()`: a parallel version of mapply()
- `future_sapply()`: a parallel version of sapply()
- `future_tapply()`: a parallel version of tapply()
- `future_vapply()`: a parallel version of vapply()
- `future_Map()`: a parallel version of Map()
- `future_replicate()`: a parallel version of replicate()
- `future_.mapply()`: a parallel version of .mapply()

Reproducibility is part of the core design, which means that perfect, parallel random number generation (RNG) is supported regardless of the amount of chunking, type of load balancing, and future backend being used.

Since these `future_*()` functions have the same arguments as the corresponding base R function, start using them is often as simple as renaming the function in the code. For example, after attaching the package:

```
library(future.apply)
```

code such as:

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
y <- lapply(x, quantile, probs = 1:3/4)
```

can be updated to:

```
y <- future_lapply(x, quantile, probs = 1:3/4)
```

The default settings in the **future** framework is to process code *sequentially*. To run the above in parallel on the local machine (on any operating system), use:

```
plan(multisession)
```

first. That's it!

To go back to sequential processing, use `plan(sequential)`. If you have access to multiple machines on your local network, use:

```
plan(cluster, workers = c("n1", "n2", "n2", "n3"))
```

This will set up four workers, one on `n1` and `n3`, and two on `n2`. If you have SSH access to some remote machines, use:

```
plan(cluster, workers = c("m1.myserver.org", "m2.myserver.org))
```

See the **future** package and `future::plan()` for more examples.

The **future.batchtools** package provides support for high-performance compute (HPC) cluster schedulers such as SGE, Slurm, and TORQUE / PBS. For example,

- `plan(batchtools_slurm)`: Process via a Slurm scheduler job queue.
- `plan(batchtools_torque)`: Process via a TORQUE / PBS scheduler job queue.

This builds on top of the queuing framework that the **batchtools** package provides. For more details on backend configuration, please see the **future.batchtools** and **batchtools** packages.

These are just a few examples of parallel/distributed backend for the future ecosystem. For more alternatives, see the 'Reverse dependencies' section on the future CRAN package page.

Author(s)

Henrik Bengtsson, except for the implementations of `future_apply()`, `future_Map()`, `future_replicate()`, `future_sapply()`, and `future_tapply()`, which are adopted from the source code of the corresponding base R functions, which are licensed under GPL (>= 2) with 'The R Core Team' as the copyright holder. Because of these dependencies, the license of this package is GPL (>= 2).

See Also

Useful links:

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

---

`future.apply.options` *Options used for future.apply*

Description

Below are the R options and environment variables that are used by the **future.apply** 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.

Details

For settings specific to the **future** package, see future::future.options page.

Options for debugging future.apply

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

Environment variables that set R options

All of the above R '`future.apply.*`' options can be set by corresponding environment variable `R_FUTURE_APPLY_*` *when the* **future.apply** *package is loaded*. For example, if `R_FUTURE_APPLY_DEBUG=TRUE`, then option '`future.apply.debug`' is set to `TRUE` (logical).

See Also

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

Examples

```
## Not run:
options(future.apply.debug = TRUE)

## End(Not run)
```

future_apply *Apply Functions Over Array Margins via Futures*

Description

future_apply() implements base::apply() using future with perfect replication of results, regardless of future backend used. It returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Usage

```
future_apply(
  X,
  MARGIN,
  FUN,
  ...,
  simplify = TRUE,
  future.envir = parent.frame(),
  future.stdout = TRUE,
  future.conditions = "condition",
  future.globals = TRUE,
  future.packages = NULL,
  future.seed = FALSE,
  future.scheduling = 1,
  future.chunk.size = NULL,
  future.label = "future_apply-%d"
)
```

Arguments

X	an array, including a matrix.
MARGIN	A vector giving the subscripts which the function will be applied over. For example, for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
FUN	A function taking at least one argument.
simplify	a logical indicating whether results should be simplified if possible.
future.envir	An environment passed as argument envir to future::future() as-is.
future.stdout	If TRUE (default), then the standard output of the underlying futures is captured, and re-outputted as soon as possible. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). If NA (not recommended), output is *not* intercepted.
future.conditions	A character string of conditions classes to be captured and relayed. The default is the same as the condition argument of future::Future(). To not intercept conditions, use conditions = character(0L). Errors are always relayed.

`future.globals` A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.

`future.packages` (optional) a character vector specifying packages to be attached in the R environment evaluating the future.

`future.seed` A logical or an integer (of length one or seven), or a list of `length(X)` with pre-generated random seeds. For details, see below section.

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

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

`future.label` If a character string, then each future is assigned a label `sprintf(future.label, chunk_idx)`. If TRUE, then the same as `future.label = "future_lapply-%d"`. If FALSE, no labels are assigned.

`...` (optional) Additional arguments passed to `FUN()`, except `future.*` arguments, which are passed on to `future_lapply()` used internally.

Value

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix. See `base::apply()` for details.

Author(s)

The implementations of `future_apply()` is adopted from the source code of the corresponding base R function, which is licensed under GPL (>= 2) with 'The R Core Team' as the copyright holder.

Examples

```
## ------------------------------------------------------------
## apply()
## ------------------------------------------------------------
X <- matrix(c(1:4, 1, 6:8), nrow = 2L)

Y0 <- apply(X, MARGIN = 1L, FUN = table)
Y1 <- future_apply(X, MARGIN = 1L, FUN = table)
print(Y1)
stopifnot(all.equal(Y1, Y0, check.attributes = FALSE)) ## FIXME

Y0 <- apply(X, MARGIN = 1L, FUN = stats::quantile)
Y1 <- future_apply(X, MARGIN = 1L, FUN = stats::quantile)
print(Y1)
```

```
stopifnot(all.equal(Y1, Y0))


## ------------------------------------------------------------
## Parallel Random Number Generation
## ------------------------------------------------------------

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

X <- matrix(c(1:4, 1, 6:8), nrow = 2L)

plan(multisession)
set.seed(0xBEEF)
Y1 <- future_apply(X, MARGIN = 1L, FUN = sample, future.seed = TRUE)
print(Y1)

plan(sequential)
set.seed(0xBEEF)
Y2 <- future_apply(X, MARGIN = 1L, FUN = sample, future.seed = TRUE)
print(Y2)

stopifnot(all.equal(Y1, Y2))
```

---

future_by *Apply a Function to a Data Frame Split by Factors via Futures*

Description

Apply a Function to a Data Frame Split by Factors via Futures

Usage

```
future_by(
  data,
  INDICES,
  FUN,
  ...,
  simplify = TRUE,
  future.envir = parent.frame()
)
```

Arguments

data	An R object, normally a data frame, possibly a matrix.
INDICES	A factor or a list of factors, each of length nrow(data).

FUN	a function to be applied to (usually data-frame) subsets of data.
simplify	logical: see base::tapply().
future.envir	An environment passed as argument envir to future::future() as-is.
...	Additional arguments pass to future_lapply() and then to FUN().

Details

Internally, data is grouped by INDICES into a list of data subset elements which is then processed by future_lapply(). When the groups differ significantly in size, the processing time may differ significantly between the groups. To correct for processing-time imbalances, adjust the amount of chunking via arguments future.scheduling and future.chunk.size.

Value

An object of class "by", giving the results for each subset. This is always a list if simplify is false, otherwise a list or array (see base::tapply()). See also base::by() for details.

Note on 'stringsAsFactors'

The future_by() is modeled as closely as possible to the behavior of base::by(). Both functions have "default" S3 methods that calls data <- as.data.frame(data) internally. This call may in turn call an S3 method for as.data.frame() that coerces strings to factors or not depending on whether it has a stringsAsFactors argument and what its default is. For example, the S3 method of as.data.frame() for lists changed its (effective) default from stringsAsFactors = TRUE to stringsAsFactors = TRUE in R 4.0.0.

Examples

```
## ------------------------------------------------------------
## by()
## ------------------------------------------------------------
library(datasets) ## warpbreaks
library(stats)    ## lm()

y0 <- by(warpbreaks, warpbreaks[,"tension"],
         function(x) lm(breaks ~ wool, data = x))

plan(multisession)
y1 <- future_by(warpbreaks, warpbreaks[,"tension"],
                function(x) lm(breaks ~ wool, data = x))

plan(sequential)
y2 <- future_by(warpbreaks, warpbreaks[,"tension"],
                function(x) lm(breaks ~ wool, data = x))
```

future_eapply *Apply a Function over a List or Vector via Futures*

Description

future_lapply() implements base::lapply() using futures with perfect replication of results, regardless of future backend used. Analogously, this is true for all the other future_nnn() functions.

Usage

```
future_eapply(
  env,
  FUN,
  ...,
  all.names = FALSE,
  USE.NAMES = TRUE,
  future.envir = parent.frame(),
  future.label = "future_eapply-%d"
)

future_lapply(
  X,
  FUN,
  ...,
  future.envir = parent.frame(),
  future.stdout = TRUE,
  future.conditions = "condition",
  future.globals = TRUE,
  future.packages = NULL,
  future.seed = FALSE,
  future.scheduling = 1,
  future.chunk.size = NULL,
  future.label = "future_lapply-%d"
)

future_replicate(
  n,
  expr,
  simplify = "array",
  future.seed = TRUE,
  ...,
  future.envir = parent.frame(),
  future.label = "future_replicate-%d"
)

future_sapply(
```

```
  X,
  FUN,
  ...,
  simplify = TRUE,
  USE.NAMES = TRUE,
  future.envir = parent.frame(),
  future.label = "future_sapply-%d"
)

future_tapply(
  X,
  INDEX,
  FUN = NULL,
  ...,
  default = NA,
  simplify = TRUE,
  future.envir = parent.frame(),
  future.label = "future_tapply-%d"
)

future_vapply(
  X,
  FUN,
  FUN.VALUE,
  ...,
  USE.NAMES = TRUE,
  future.envir = parent.frame(),
  future.label = "future_vapply-%d"
)
```

Arguments

env	An R environment.
FUN	A function taking at least one argument.
all.names	If TRUE, the function will also be applied to variables that start with a period (.), otherwise not. See base::eapply() for details.
USE.NAMES	See base::sapply().
future.envir	An environment passed as argument envir to future::future() as-is.
future.label	If a character string, then each future is assigned a label sprintf(future.label, chunk_idx). If TRUE, then the same as future.label = "future_lapply-%d". If FALSE, no labels are assigned.
X	An R object for which a split method exists. Typically vector-like, allowing subsetting with [, or a data frame.
future.stdout	If TRUE (default), then the standard output of the underlying futures is captured, and re-outputted as soon as possible. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). If NA (not recommended), output is *not* intercepted.

future.conditions
: A character string of conditions classes to be captured and relayed. The default is the same as the condition argument of future::Future(). To not intercept conditions, use conditions = character(0L). Errors are always relayed.

future.globals
: A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.

future.packages
: (optional) a character vector specifying packages to be attached in the R environment evaluating the future.

future.seed
: A logical or an integer (of length one or seven), or a list of length(X) with pre-generated random seeds. For details, see below section.

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

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

n
: The number of replicates.

expr
: An R expression to evaluate repeatedly.

simplify
: See base::sapply() and base::tapply(), respectively.

INDEX
: A list of one or more factors, each of same length as X. The elements are coerced to factors by as.factor(). Can also be a formula, which is useful if X is a data frame; see the f argument in split() for interpretation.

default
: See base::tapply().

FUN.VALUE
: A template for the required return value from each FUN(X[ii], ...). Types may be promoted to a higher type within the ordering logical < integer < double < complex, but not demoted. See base::vapply() for details.

...
: (optional) Additional arguments passed to FUN(). For future_*apply() functions and replicate(), any future.* arguments part of \ldots are passed on to future_lapply() used internally. Importantly, if this is called inside another function which also declares ... arguments, do not forget to explicitly pass such ... arguments down to the future_*apply() function too, which will then pass them on to FUN(). See below for an example.

Value

A named (unless USE.NAMES = FALSE) list. See base::eapply() for details.

For future_lapply(), a list with same length and names as X. See base::lapply() for details.

future_replicate() is a wrapper around future_sapply() and return simplified object according to the simplify argument. See base::replicate() for details. Since future_replicate() usually involves random number generation (RNG), it uses future.seed = TRUE by default in order

produce sound random numbers regardless of future backend and number of background workers used.

For `future_sapply()`, a vector with same length and names as `X`. See `base::sapply()` for details.

`future_tapply()` returns an array with mode `"list"`, unless `simplify = TRUE` (default) *and* `FUN` returns a scalar, in which case the mode of the array is the same as the returned scalars. See `base::tapply()` for details.

For `future_vapply()`, a vector with same length and names as `X`. See `base::vapply()` for details.

Global variables

Argument `future.globals` may be used to control how globals should be handled similarly how the `globals` argument is used with `future()`. Since all function calls use the same set of globals, this function can do any gathering of globals upfront (once), which is more efficient than if it would be done for each future independently. If `TRUE` (default), then globals are automatically identified and gathered. If a character vector of names is specified, then those globals are gathered. If a named list, then those globals are used as is. In all cases, `FUN` and any `\ldots` arguments are automatically passed as globals to each future created as they are always needed.

Reproducible random number generation (RNG)

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

RNG reproducibility is achieved by pregenerating the random seeds for all iterations (over `X`) by using parallel RNG streams. In each iteration, these seeds are set before calling `FUN(X[[ii]], ...)`. *Note, for large* `length(X)` *this may introduce a large overhead.*

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

If `future.seed = FALSE`, it is expected that none of the `FUN(X[[ii]], ...)` function calls 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 `future.seed = NULL`, is like `future.seed = FALSE` but without the check whether random numbers were generated or not.

As input, `future.seed` may also take a fixed initial seed (integer), either as a full parallel RNG seed, or as a seed generating such a full parallel seed. This seed will be used to generated `length(X)` parallel RNG streams.

In addition to the above, it is possible to specify a pre-generated sequence of RNG seeds as a list such that `length(future.seed) == length(X)` and where each element is an integer seed vector that can be assigned to `.Random.seed`. One approach to generate a set of valid RNG seeds based on fixed initial seed (here `42L`) is:

```
seeds <- future_lapply(seq_along(X), FUN = function(x) .Random.seed,
                       future.chunk.size = Inf, future.seed = 42L)
```

Note that `as.list(seq_along(X))` **is *not* a valid set of such** `.Random.seed` **values.**

In all cases but `future.seed = FALSE` and `NULL`, the RNG state of the calling R processes after this function returns is guaranteed to be "forwarded one step" from the RNG state that was before the

call and in the same way regardless of future.seed, future.scheduling and future strategy used. This is done in order to guarantee that an R script calling future_lapply() multiple times should be numerically reproducible given the same initial seed.

Load balancing ("chunking")

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

The value future.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 future.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.

Control processing order of elements

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

For example, future.scheduling = structure(TRUE, ordering = "random"). *Note*, when elements are processed out of order, then captured standard output and conditions are relayed in that order as well.

Author(s)

The implementations of future_replicate(), future_sapply(), and future_tapply() are adopted from the source code of the corresponding base R functions, which are licensed under GPL (>= 2) with 'The R Core Team' as the copyright holder.

Examples

```
## ------------------------------------------------------------
## lapply(), sapply(), tapply()
## ------------------------------------------------------------
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE, FALSE, FALSE, TRUE))
y0 <- lapply(x, FUN = quantile, probs = 1:3/4)
y1 <- future_lapply(x, FUN = quantile, probs = 1:3/4)
print(y1)
stopifnot(all.equal(y1, y0))
```

```
y0 <- sapply(x, FUN = quantile)
y1 <- future_sapply(x, FUN = quantile)
print(y1)
stopifnot(all.equal(y1, y0))

y0 <- vapply(x, FUN = quantile, FUN.VALUE = double(5L))
y1 <- future_vapply(x, FUN = quantile, FUN.VALUE = double(5L))
print(y1)
stopifnot(all.equal(y1, y0))


## ------------------------------------------------------------
## Parallel Random Number Generation
## ------------------------------------------------------------

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

plan(multisession)
set.seed(0xBEEF)
y1 <- future_lapply(1:5, FUN = rnorm, future.seed = TRUE)
str(y1)

plan(sequential)
set.seed(0xBEEF)
y2 <- future_lapply(1:5, FUN = rnorm, future.seed = TRUE)
str(y2)

stopifnot(all.equal(y1, y2))



## ------------------------------------------------------------
## Process chunks of data.frame rows in parallel
## ------------------------------------------------------------
iris <- datasets::iris
chunks <- split(iris, seq(1, nrow(iris), length.out = 3L))
y0 <- lapply(chunks, FUN = function(iris) sum(iris$Sepal.Length))
y0 <- do.call(sum, y0)
y1 <- future_lapply(chunks, FUN = function(iris) sum(iris$Sepal.Length))
y1 <- do.call(sum, y1)
print(y1)
stopifnot(all.equal(y1, y0))


## ------------------------------------------------------------
## Remember to pass down '...' arguments
## ------------------------------------------------------------
## It is important that we don't use '...' as a global variable,
## as attempted in the following not_okay_fcn()
bad_fcn <- function(X, ...) {
  y <- future_lapply(X, FUN = function(x) {
    mean(x, ...)  ## here '...' is a global variable
```

```
  })
  y
}

## Instead, make sure to pass '...' via arguments all the way through
good_fcn <- function(X, ...) { ## outer '...'
  y <- future_lapply(X, FUN = function(x, ...) {
    mean(x, ...)  ## here '...' is an argument of FUN()
  }, ...) ## pass outer '...' to FUN()
  y
}
```

---

future_Filter *Apply a Function to Multiple List or Vector Arguments*

Description

future_mapply() implements base::mapply() using futures with perfect replication of results, regardless of future backend used. Analogously to mapply(), future_mapply() is a multivariate version of future_sapply(). It applies FUN to the first elements of each \ldots argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

Usage

```
future_Filter(f, x, ...)

future_Map(
  f,
  ...,
  future.envir = parent.frame(),
  future.label = "future_Map-%d"
)

future_mapply(
  FUN,
  ...,
  MoreArgs = NULL,
  SIMPLIFY = TRUE,
  USE.NAMES = TRUE,
  future.envir = parent.frame(),
  future.stdout = TRUE,
  future.conditions = "condition",
  future.globals = TRUE,
  future.packages = NULL,
  future.seed = FALSE,
```

```
  future.scheduling = 1,
  future.chunk.size = NULL,
  future.label = "future_mapply-%d"
)

future_.mapply(FUN, dots, MoreArgs, ..., future.label = "future_.mapply-%d")
```

Arguments

f	A function of the arity k if future_Map() is called with k arguments.
x	A vector-like object to iterate over.
future.envir	An environment passed as argument envir to future::future() as-is.
future.label	If a character string, then each future is assigned a label sprintf(future.label, chunk_idx). If TRUE, then the same as future.label = "future_lapply-%d". If FALSE, no labels are assigned.
FUN	A function to apply, found via base::match.fun().
MoreArgs	A list of other arguments to FUN.
SIMPLIFY	A logical or character string; attempt to reduce the result to a vector, matrix or higher dimensional array; see the simplify argument of base::sapply().
USE.NAMES	A logical; use names if the first \ldots argument has names, or if it is a character vector, use that character vector as the names.
future.stdout	If TRUE (default), then the standard output of the underlying futures is captured, and re-outputted as soon as possible. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). If NA (not recommended), output is *not* intercepted.
future.conditions	A character string of conditions classes to be captured and relayed. The default is the same as the condition argument of future::Future(). To not intercept conditions, use conditions = character(0L). Errors are always relayed.
future.globals	A logical, a character vector, or a named list for controlling how globals are handled. For details, see future_lapply().
future.packages	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
future.seed	A logical or an integer (of length one or seven), or a list of max(lengths(list(...))) with pre-generated random seeds. For details, see future_lapply().
future.scheduling	Average number of futures ("chunks") per worker. If 0.0, then a single future is used to process all elements of X. 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 in X). If Inf or FALSE, then one future per element of X is used. Only used if future.chunk.size is NULL.
future.chunk.size	The average number of elements per future ("chunk"). If Inf, then all elements are processed in a single future. If NULL, then argument future.scheduling is used.

dots	A list of arguments to vectorize over (vectors or lists of strictly positive length, or all of zero length).
...	Arguments to vectorize over, will be recycled to common length, or zero if one of them is of length zero.

Details

Note that base::.mapply(), which future_.mapply() is modeled after is listed as an "internal" function in R despite being exported.

Value

See base::Filter() for details.

future_Map() is a simple wrapper to future_mapply() which does not attempt to simplify the result. See base::Map() for details.

future_mapply() returns a list, or for SIMPLIFY = TRUE, a vector, array or list. See base::mapply() for details.

future_.mapply() returns a list. See base::.mapply() for details.

Author(s)

The implementations of future_Filter() is adopted from the source code of the corresponding base R function Filter(), which is licensed under GPL (>= 2) with 'The R Core Team' as the copyright holder.

The implementations of future_Map() is adopted from the source code of the corresponding base R function Map(), which is licensed under GPL (>= 2) with 'The R Core Team' as the copyright holder.

Examples

```
## ------------------------------------------------------------
## Filter()
## ------------------------------------------------------------
is_even <- function(x) { x %% 2 == 0 }
x <- sample.int(100, size = 1000, replace = TRUE)
y <- future_Filter(is_even, x)


## ------------------------------------------------------------
## mapply()
## ------------------------------------------------------------
y0 <- mapply(rep, 1:4, 4:1)
y1 <- future_mapply(rep, 1:4, 4:1)
stopifnot(identical(y1, y0))

y0 <- mapply(rep, times = 1:4, x = 4:1)
y1 <- future_mapply(rep, times = 1:4, x = 4:1)
stopifnot(identical(y1, y0))
```

```
y0 <- mapply(rep, times = 1:4, MoreArgs = list(x = 42))
y1 <- future_mapply(rep, times = 1:4, MoreArgs = list(x = 42))
stopifnot(identical(y1, y0))

y0 <- mapply(function(x, y) seq_len(x) + y,
             c(a =  1, b = 2, c = 3),  # names from first
             c(A = 10, B = 0, C = -10))
y1 <- future_mapply(function(x, y) seq_len(x) + y,
                    c(a =  1, b = 2, c = 3),  # names from first
                    c(A = 10, B = 0, C = -10))
stopifnot(identical(y1, y0))

word <- function(C, k) paste(rep.int(C, k), collapse = "")
y0 <- mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE)
y1 <- future_mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE)
stopifnot(identical(y1, y0))


## ------------------------------------------------------------
## Parallel Random Number Generation
## ------------------------------------------------------------

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

plan(multisession)
set.seed(0xBEEF)
y1 <- future_mapply(stats::runif, n = 1:4, max = 2:5,
                    MoreArgs = list(min = 1), future.seed = TRUE)
print(y1)

plan(sequential)
set.seed(0xBEEF)
y2 <- future_mapply(stats::runif, n = 1:4, max = 2:5,
                    MoreArgs = list(min = 1), future.seed = TRUE)
print(y2)

stopifnot(all.equal(y1, y2))
```

`future_kernapply` *Apply Smoothing Kernel in Parallel*

Description

`future_kernapply()` is a futurized version of `stats::kernapply()`, i.e. it computes, in parallel, the convolution between an input sequence and a specific kernel. Parallelization takes place over columns when `x` is a matrix, including a `ts` matrix.

Usage

```
future_kernapply(x, ...)

## Default S3 method:
future_kernapply(x, k, circular = FALSE, ...)

## S3 method for class 'ts'
future_kernapply(x, k, circular = FALSE, ...)
```

Arguments

x	an input vector, matrix, time series or kernel to be smoothed.
...	arguments passed to or from other methods.
k	smoothing "tskernel" object.
circular	a logical indicating whether the input sequence to be smoothed is treated as circular, i.e., periodic.

Value

See stats::kernapply() for details.

Examples

```
library(datasets)
library(stats)

X <- EuStockMarkets[, 1:2]
k <- kernel("daniell", 50)  # a long moving average
X_smooth <- future_kernapply(X, k = k)
```

Index