Package ‘future.apply’

January 17, 2019

Version 1.1.0

Title Apply Function to Elements in Parallel using Futures

Depends R (>= 3.2.0), future (>= 1.10.0)

Imports globals (>= 0.12.4)

Suggests datasets, stats, tools, listenv (>= 0.7.0), R.rsp, markdown

VignetteBuilder R.rsp

Description Implementations of apply(), eapply(), lapply(), Map(), 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.

License GPL (>= 2)

LazyLoad TRUE

URL `https://github.com/HenrikBengtsson/future.apply`

BugReports `https://github.com/HenrikBengtsson/future.apply/issues`

RoxygenNote 6.1.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph],
R Core Team [cph, ctb]

Maintainer Henrik Bengtsson <`henrikb@braju.com`>

Repository CRAN

Date/Publication 2019-01-17 05:40:03 UTC

R topics documented:

future.apply 2
future_apply 3
future_eapply 5
future_Map 9

Index **12**

`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_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()

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(multiprocess)
```

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).

`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, ...)
```

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.

... (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(multiprocess)
Y1 <- future_apply(X, MARGIN = 1L, FUN = sample, future.seed = 0xBEEF)
```

```
print(Y1)

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

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

`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_lapply(X, FUN, ..., future.stdout = TRUE,
  future.conditions = c("message", "warning"), future.globals = TRUE,
  future.packages = NULL, future.lazy = FALSE, future.seed = FALSE,
  future.scheduling = 1, future.chunk.size = NULL)

future_replicate(n, expr, simplify = "array", future.seed = TRUE, ...)

future_sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

future_tapply(X, INDEX, FUN = NULL, ..., default = NA,
  simplify = TRUE)

future_vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

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()`.
`X`	A vector-like object to iterate over.

`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 to relay messages and warnings. 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.lazy` Specifies whether the futures should be resolved lazily or eagerly (default).

`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()`. See also `base::tapply()`.

`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 `...` are passed on to `future_lapply()` used internally.

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, NULL or not is specified (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 ... arguments are automatically passed as globals to each future created as they are always needed.

Reproducible random number generation (RNG)

Unless future.seed = FALSE, 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 L'Ecuyer-CMRG 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.* As input (future.seed), a fixed seed (integer) may be given, either as a full L'Ecuyer-CMRG RNG seed (vector of 1+6 integers) or as a seed generating such a full L'Ecuyer-CMRG seed. If future.seed = TRUE, then .Random.seed is returned if it holds a L'Ecuyer-CMRG RNG seed, otherwise one is created randomly. If future.seed = NA, a L'Ecuyer-CMRG RNG seed is randomly created. If none of the function calls FUN(X[[ii]], ...) uses random number generation, then future.seed = FALSE may be used.

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, 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.

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 much 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").

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(multiprocess)
y1 <- future_lapply(1:5, FUN = rnorm, future.seed = 0xBEEF)
str(y1)
```

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

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

future_Map *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 ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

Usage

```
future_Map(f, ...)

future_mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
  USE.NAMES = TRUE, future.stdout = TRUE,
  future.conditions = c("message", "warning"), future.globals = TRUE,
  future.packages = NULL, future.lazy = FALSE, future.seed = FALSE,
  future.scheduling = 1, future.chunk.size = NULL)
```

Arguments

f A function of the arity k if future_Map() is called with k arguments.

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 ... 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 to relay messages and warnings. 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.lazy`
: Specifies whether the futures should be resolved lazily or eagerly (default).

`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.

`...`
: Arguments to vectorize over (vectors or lists of strictly positive length, or all of zero length).

Value

`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.

Author(s)

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

```
## ---------------------------------------------------------
## 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(multiprocess)
y1 <- future_mapply(stats::runif, n = 1:4, max = 2:5,
                    MoreArgs = list(min = 1), future.seed = 0xBEEF)
print(y1)

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

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

Index

∗Topic **iteration**
    future.apply, 2
    future_eapply, 5
    future_Map, 9
∗Topic **manip**
    future.apply, 2
    future_eapply, 5
    future_Map, 9
∗Topic **programming**
    future.apply, 2
    future_eapply, 5
    future_Map, 9
.Random.seed, *7*

apply(), *2*

base::apply(), *3*, *4*
base::eapply(), *5*, *6*
base::lapply(), *5*, *6*
base::Map(), *10*
base::mapply(), *9*, *10*
base::match.fun(), *9*
base::replicate(), *6*
base::sapply(), *5–7*, *9*
base::tapply(), *6*, *7*
base::vapply(), *6*, *7*

eapply(), *2*

future.apply, 2
future.apply-package *(future.apply)*, 2
future::plan(), *3*
future_apply, 3
future_apply(), *2*
future_eapply, 5
future_eapply(), *2*
future_lapply *(future_eapply)*, 5
future_lapply(), *2*, *9*, *10*
future_Map, 9
future_Map(), *2*
future_mapply *(future_Map)*, 9
future_mapply(), *2*
future_replicate *(future_eapply)*, 5
future_replicate(), *2*
future_sapply *(future_eapply)*, 5
future_sapply(), *2*
future_tapply *(future_eapply)*, 5
future_tapply(), *2*
future_vapply *(future_eapply)*, 5
future_vapply(), *2*

lapply(), *2*

Map(), *2*
mapply(), *2*

replicate(), *2*

sapply(), *2*

tapply(), *2*

vapply(), *2*