Package 'future.apply'

May 1, 2018

Version 0.2.0

Title Apply Function to Elements in Parallel using Futures

Depends R (>= 3.2.0), future (>= 1.8.0)

Imports globals (>= 0.11.0)

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

VignetteBuilder R.rsp

Description Implementations of lapply(), sapply(), tapply(), vapply() and friends 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 LGPL (>= 2.1)

LazyLoad TRUE

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

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

RoxygenNote 6.0.1

NeedsCompilation no

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

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2018-05-01 05:49:21 UTC

R topics documented:

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_eapply()`: a parallel version of eapply()
- `future_lapply()`: a parallel version of lapply()
- `future_sapply()`: a parallel version of sapply()
- `future_tapply()`: a parallel version of tapply()
- `future_vapply()`: a parallel version of vapply()
- `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.

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

Description

Apply a Function over a List or Vector via Futures

Usage

```
future_eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)

future_lapply(X, FUN, ..., future.globals = TRUE, future.packages = NULL,
  future.seed = FALSE, future.lazy = FALSE, future.scheduling = 1)

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.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.lazy	Specifies whether the futures should be resolved lazily or eagerly (default).
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.
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(), anyfuture.*arguments part of ... are passed on tofuture_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. Use this alternative with caution. **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.

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

```
## 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))
```

Index