Package 'doFuture'

October 13, 2022

Version 0.12.2

Title A Universal Foreach Parallel Adapter using the Future API of the
'future' Package

Depends foreach (>= 1.5.0), future (>= 1.22.1)

Imports globals, iterators, parallel, utils

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

VignetteBuilder R.rsp

Description Provides a '%dopar%' adapter such that any type of futures can
be used as backends for the 'foreach' framework.

License LGPL (>= 2.1)

LazyLoad TRUE

URL <https://doFuture.futureverse.org>,
<https://github.com/HenrikBengtsson/doFuture>

BugReports <https://github.com/HenrikBengtsson/doFuture/issues>

RoxygenNote 7.1.2

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2022-04-26 09:50:02 UTC

R topics documented:

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

Description

The **doFuture** package provides a %dopar% adapter for the **foreach** package such that *any* type of future (that is supported by Future API of the **future** package) can be used for asynchronous (parallel/distributed) or synchronous (sequential) processing.

Details

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.

Usage

To use futures with the **foreach** package, load **doFuture**, use registerDoFuture() to register it to be used as a %dopar% adapter (no need to ever use %do%). After this, how and where the computations are performed is controlled solely by the future strategy set, which in controlled by future::plan(). For example:

- plan(multisession): multiple R processes on the local machine.
- plan(cluster, workers = c("n1", "n2", "n2", "n3")): multiple R processes on external machines.

See the **future** package for more examples.

Built-in backends

The built-in backends of **doFuture** are for instance 'multicore' (forked processes), 'multisession' (background R sessions), and ad-hoc 'cluster' (background R sessions on local and / or remote machines). Additional futures are provided by other "future" packages (see below for some examples).

Backends for high-performance compute clusters

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.

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 doFuture::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* 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** package 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. For valid parallel RNG, it is recommended to use the **doRNG** package, where the %dorng% operator is used in place of %dopar%. Note that **doRNG** is designed to work with any type of foreach adapter including **doFuture**.

Examples

```
library(doFuture)
registerDoFuture()
plan(multisession)
library(iterators)  # iter()


## 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))
```

registerDoFuture	*Registers the future %dopar% backend*

Description

Register the doFuture parallel adapter to be used by the **foreach** package.

Usage

```
registerDoFuture()
```

Value

Invisibly returns the previously registered foreach backend.

For package developers

Please refrain from modifying the foreach backend inside your packages / functions, i.e. do not call `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 done using:

```
 oldDoPar <- registerDoFuture()
on.exit(with(oldDoPar, foreach::setDoPar(fun=fun, data=data, info=info)), add = 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.*

Examples

```
registerDoFuture()
```

Index