Package 'future'

May 9, 2019

Version 1.13.0

Title Unified Parallel and Distributed Processing in R for Everyone

Imports digest, globals (>= 0.12.4), listenv (>= 0.7.0), parallel, utils

Suggests R.rsp, markdown

VignetteBuilder R.rsp

Description The purpose of this package is to provide a lightweight and unified Future API for sequential and parallel processing of R expression via futures. The simplest way to evaluate an expression in parallel is to use `x %<-% { expression }` with `plan(multiprocess)`. This package implements sequential, multicore, multisession, and cluster futures. With these, R expressions can be evaluated on the local machine, in parallel a set of local machines, or distributed on a mix of local and remote machines.
Extensions to this package implement additional backends for processing futures via compute cluster schedulers etc.
Because of its unified API, there is no need to modify any code in order switch from sequential on the local machine to, say, distributed processing on a remote compute cluster.
Another strength of this package is that global variables and functions are automatically identified and exported as needed, making it straightforward to tweak existing code to make use of futures.

License LGPL (>= 2.1)

LazyLoad TRUE

ByteCompile TRUE

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

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

RoxygenNote 6.1.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2019-05-08 22:50:03 UTC

R topics documented:

`backtrace` *Back trace the expressions evaluated when an error was caught*

Description

Back trace the expressions evaluated when an error was caught

Usage

```
backtrace(future, envir = parent.frame(), ...)
```

Arguments

future	A future with a caught error.
envir	the environment where to locate the future.
...	Not used.

Value

A @list with the future's call stack that led up to the error.

Examples

```
my_log <- function(x) log(x)
foo <- function(...) my_log(...)

f <- future({ foo("a") })
res <- tryCatch({
  v <- value(f)
}, error = function(ex) {
  t <- backtrace(f)
  print(t)
})
```

---

cluster *Create a cluster future whose value will be resolved asynchronously in a parallel process*

Description

A cluster future is a future that uses cluster evaluation, which means that its *value is computed and resolved in parallel in another process*.

Usage

```
cluster(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE, persistent = FALSE,
  workers = availableWorkers(), user = NULL, revtunnel = TRUE,
  homogeneous = TRUE, gc = FALSE, earlySignal = FALSE,
  label = NULL, ...)
```

Arguments

expr	An R expression.
envir	The environment from where global objects should be identified.
substitute	If TRUE, argument expr is substitute():ed, otherwise not.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.

seed	(optional) A L'Ecuyer-CMRG RNG seed.
globals	(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()`.
persistent	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
workers	A `cluster` object, a character vector of host names, a positive numeric scalar, or a function. If a character vector or a numeric scalar, a `cluster` object is created using `makeClusterPSOCK(workers)`. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return any of the above types.
user	(optional) The user name to be used when communicating with another host.
revtunnel	If TRUE, reverse SSH tunneling is used for the PSOCK cluster nodes to connect back to the master R process. This avoids the hassle of firewalls, port forwarding and having to know the internal / public IP address of the master R session.
homogeneous	If TRUE, all cluster nodes is assumed to use the same path to '`Rscript`' as the main R session. If FALSE, the it is assumed to be on the PATH for each node.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) only after the value of the future is collected. Exactly when the values are collected may depend on various factors such as number of free workers and whether `earlySignal` is TRUE (more frequently) or FALSE (less frequently). *Some types of futures ignore this argument.*
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
label	An optional character string label attached to the future.
...	Additional named elements passed to `ClusterFuture()`.

Details

This function will block if all available R cluster nodes are occupied and will be unblocked as soon as one of the already running cluster futures is resolved.

The preferred way to create an cluster future is not to call this function directly, but to register it via `plan(cluster)` such that it becomes the default mechanism for all futures. After this `future()` and `%<-%` will create *cluster futures*.

Value

A ClusterFuture.

Examples

```
## Use cluster futures
cl <- parallel::makeCluster(2L, timeout = 60)
plan(cluster, workers = cl)
```

```
## A global variable
a <- 0

## Create multicore future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A cluster future is evaluated in a separate process.
## Changing the value of a global variable will not
## affect the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)

## CLEANUP
parallel::stopCluster(cl)
```

`future` *Create a future*

Description

Creates a future that evaluates an R expression or a future that calls an R function with a set of arguments. How, when, and where these futures are evaluated can be configured using `plan()` such that it is evaluated in parallel on, for instance, the current machine, on a remote machine, or via a job queue on a compute cluster. Importantly, any R code using futures remains the same regardless on these settings and there is no need to modify the code when switching from, say, sequential to parallel processing.

Usage

```
future(expr, envir = parent.frame(), substitute = TRUE,
  globals = TRUE, packages = NULL, lazy = FALSE, seed = NULL,
  evaluator = plan("next"), ...)

futureAssign(x, value, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE, ...,
  assign.env = envir)

x %<-% value
```

```
futureCall(FUN, args = list(), envir = parent.frame(), lazy = FALSE,
  seed = NULL, globals = TRUE, packages = NULL,
  evaluator = plan("next"), ...)
```

Arguments

expr, value	An R expression.
envir	The environment from where global objects should be identified.
substitute	If TRUE, argument expr is substitute():ed, otherwise not.
globals	(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	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
seed	(optional) A L'Ecuyer-CMRG RNG seed.
evaluator, ...	(internal) The actual function that evaluates the future expression and returns a Future and additional arguments passed to it. The evaluator function should accept all of the same arguments as the ones listed here (except evaluator, FUN and args). The default evaluator function is the one that the user has specified via plan().
x	the name of a future variable, which will hold the value of the future expression (as a promise).
assign.env	The environment to which the variable should be assigned.
FUN	A function to be evaluated.
args	A list of arguments passed to function FUN.

Details

The state of a future is either unresolved or resolved. The value of a future can be retrieved using `v <- value(f)`. Querying the value of a non-resolved future will *block* the call until the future is resolved. It is possible to check whether a future is resolved or not without blocking by using `resolved(f)`.

For a future created via a future assignment (`x %<-% value` or `futureAssign("x", value)`), the value is bound to a promise, which when queried will internally call `value()` on the future and which will then be resolved into a regular variable bound to that value. For example, with future assignment `x %<-% value`, the first time variable `x` is queried the call blocks if (and only if) the future is not yet resolved. As soon as it is resolved, and any succeeding queries, querying `x` will immediately give the value.

The future assignment construct `x %<-% value` is not a formal assignment per se, but a binary infix operator on objects `x` and expression `value`. However, by using non-standard evaluation, this constructs can emulate an assignment operator similar to `x <- value`. Due to R's precedence rules of operators, future expressions often need to be explicitly bracketed, e.g. `x %<-% { a + b }`.

The futureCall() function works analogously to do.call(), which calls a function with a set of arguments. The difference is that do.call() returns the value of the call whereas futureCall() returns a future.

Value

f <- future(expr) creates a Future f that evaluates expression expr, the value of the future is retrieved using v <- value(f).

x %<-% value (a future assignment) and futureAssign("x", value) create a Future that evaluates expression expr and binds its value (as a promise) to a variable x. The value of the future is automatically retrieved when the assigned variable (promise) is queried. The future itself is returned invisibly, e.g. f <- futureAssign("x", expr) and f <- (x %<-% expr). Alternatively, the future of a future variable x can be retrieved without blocking using f <- futureOf(x). Both the future and the variable (promise) are assigned to environment assign.env where the name of the future is .future_<name>.

f <- futureCall(FUN, args) creates a Future f that calls function FUN with arguments args, where the value of the future is retrieved using x <- value(f).

Eager or lazy evaluation

By default, a future is resolved using *eager* evaluation (lazy = FALSE). This means that the expression starts to be evaluated as soon as the future is created.

As an alternative, the future can be resolved using *lazy* evaluation (lazy = TRUE). This means that the expression will only be evaluated when the value of the future is requested. *Note that this means that the expression may not be evaluated at all - it is guaranteed to be evaluated if the value is requested.*

For future assignments, lazy evaluation can be controlled via the %lazy% operator, e.g. x %<-% { expr } %lazy% TRUE.

Globals used by future expressions

Global objects (short *globals*) are objects (e.g. variables and functions) that are needed in order for the future expression to be evaluated while not being local objects that are defined by the future expression. For example, in

```
a <- 42
f <- future({ b <- 2; a * b })
```

variable a is a global of future assignment f whereas b is a local variable. In order for the future to be resolved successfully (and correctly), all globals need to be gathered when the future is created such that they are available whenever and wherever the future is resolved.

The default behavior (globals = TRUE) of all evaluator functions, is that globals are automatically identified and gathered. More precisely, globals are identified via code inspection of the future expression expr and their values are retrieved with environment envir as the starting point (basically via get(global, envir = envir, inherits = TRUE)). *In most cases, such automatic collection of globals is sufficient and less tedious and error prone than if they are manually specified.*

However, for full control, it is also possible to explicitly specify exactly which the globals are by providing their names as a character vector. In the above example, we could use

```
a <- 42
f <- future({ b <- 2; a * b }, globals = "a")
```

Yet another alternative is to explicitly specify also their values using a named list as in

```
a <- 42
f <- future({ b <- 2; a * b }, globals = list(a = a))
```

or

```
f <- future({ b <- 2; a * b }, globals = list(a = 42))
```

Specifying globals explicitly avoids the overhead added from automatically identifying the globals and gathering their values. Furthermore, if we know that the future expression does not make use of any global variables, we can disable the automatic search for globals by using

```
f <- future({ a <- 42; b <- 2; a * b }, globals = FALSE)
```

Future expressions often make use of functions from one or more packages. As long as these functions are part of the set of globals, the future package will make sure that those packages are attached when the future is resolved. Because there is no need for such globals to be frozen or exported, the future package will not export them, which reduces the amount of transferred objects. For example, in

```
x <- rnorm(1000)
f <- future({ median(x) })
```

variable `x` and `median()` are globals, but only `x` is exported whereas `median()`, which is part of the **stats** package, is not exported. Instead it is made sure that the **stats** package is on the search path when the future expression is evaluated. Effectively, the above becomes

```
x <- rnorm(1000)
f <- future({
  library("stats")
  median(x)
})
```

To manually specify this, one can either do

```
x <- rnorm(1000)
f <- future({
  median(x)
}, globals = list(x = x, median = stats::median)
```

or

```
x <- rnorm(1000)
f <- future({
  library("stats")
  median(x)
}, globals = list(x = x))
```

Both are effectively the same.

Although rarely needed, a combination of automatic identification and manual specification of globals is supported via attributes `add` (to add false negatives) and `ignore` (to ignore false positives) on value `TRUE`. For example, with `globals = structure(TRUE, ignore = "b", add = "a")` any globals automatically identified except `b` will be used in addition to global `a`.

When using future assignments, globals can be specified analogously using the `%globals%` operator, e.g.

```
x <- rnorm(1000)
y %<-% { median(x) } %globals% list(x = x, median = stats::median)
```

See Also

How, when and where futures are resolved is given by the *future strategy*, which can be set by the end user using the `plan()` function. The future strategy must not be set by the developer, e.g. it must not be called within a package.

Examples

```
## Evaluate futures in parallel
plan(multiprocess)

## Data
x <- rnorm(100)
y <- 2 * x + 0.2 + rnorm(100)
w <- 1 + x ^ 2


## EXAMPLE: Regular assignments (evaluated sequentially)
fitA <- lm(y ~ x, weights = w)      ## with offset
fitB <- lm(y ~ x - 1, weights = w)  ## without offset
fitC <- {
  w <- 1 + abs(x)  ## Different weights
  lm(y ~ x, weights = w)
}
print(fitA)
print(fitB)
print(fitC)


## EXAMPLE: Future assignments (evaluated in parallel)
fitA %<-% lm(y ~ x, weights = w)      ## with offset
fitB %<-% lm(y ~ x - 1, weights = w)  ## without offset
fitC %<-% {
  w <- 1 + abs(x)
  lm(y ~ x, weights = w)
}
print(fitA)
print(fitB)
print(fitC)
```

```
## EXAMPLE: Explicitly create futures (evaluated in parallel)
## and retrieve their values
fA <- future( lm(y ~ x, weights = w) )
fB <- future( lm(y ~ x - 1, weights = w) )
fC <- future({
  w <- 1 + abs(x)
  lm(y ~ x, weights = w)
})
fitA <- value(fA)
fitB <- value(fB)
fitC <- value(fC)
print(fitA)
print(fitB)
print(fitC)


## EXAMPLE: futureCall() and do.call()
x <- 1:100
y0 <- do.call(sum, args = list(x))
print(y0)

f1 <- futureCall(sum, args = list(x))
y1 <- value(f1)
print(y1)
```

`futureOf` *Get the future of a future variable*

Description

Get the future of a future variable that has been created directly or indirectly via `future()`.

Usage

```
futureOf(var = NULL, envir = parent.frame(), mustExist = TRUE,
  default = NA, drop = FALSE)
```

Arguments

`var`	the variable. If NULL, all futures in the environment are returned.
`envir`	the environment where to search from.
`mustExist`	If TRUE and the variable does not exists, then an informative error is thrown, otherwise NA is returned.
`default`	the default value if future was not found.
`drop`	if TRUE and `var` is NULL, then returned list only contains futures, otherwise also `default` values.

Value

A Future (or default). If var is NULL, then a named list of Future:s are returned.

Examples

```
a %<-% { 1 }

f <- futureOf(a)
print(f)

b %<-% { 2 }

f <- futureOf(b)
print(f)

## All futures
fs <- futureOf()
print(fs)


## Futures part of environment
env <- new.env()
env$c %<-% { 3 }

f <- futureOf(env$c)
print(f)

f2 <- futureOf(c, envir = env)
print(f2)

f3 <- futureOf("c", envir = env)
print(f3)

fs <- futureOf(envir = env)
print(fs)
```

---

futures *Get all futures in a container*

Description

Gets all futures in an environment, a list, or a list environment and returns an object of the same class (and dimensions). Non-future elements are returned as is.

Usage

```
futures(x, ...)
```

Arguments

x	An environment, a list, or a list environment.
...	Not used.

Details

This function is useful for retrieve futures that were created via future assignments (%<-%) and therefore stored as promises. This function turns such promises into standard Future objects.

Value

An object of same type as x and with the same names and/or dimensions, if set.

---

makeClusterMPI *Create a Message Passing Interface (MPI) cluster of* R *workers for parallel processing*

Description

The makeClusterMPI() function creates an MPI cluster of R workers for parallel processing. This function utilizes makeCluster(..., type = "MPI") of the **parallel** package and tweaks the cluster in an attempt to avoid stopCluster() from hanging [1]. *WARNING: This function is very much in a beta version and should only be used if* parallel::makeCluster(..., type = "MPI") *fails.*

Usage

```
makeClusterMPI(workers, ..., autoStop = FALSE,
  verbose = getOption("future.debug", FALSE))
```

Arguments

workers	The number workers (as a positive integer).
...	Optional arguments passed to makeCluster(workers, type = "MPI", ...).
autoStop	If TRUE, the cluster will be automatically stopped
verbose	If TRUE, informative messages are outputted.

Details

Creating MPI clusters requires the ***Rmpi*** *package.*

Value

An object of class "FutureMPIcluster" consisting of a list of "MPInode" workers.

References

[1] R-sig-hpc thread Rmpi: mpi.close.Rslaves() 'hangs' on 2017-09-28.

See Also

`makeClusterPSOCK()` and `parallel::makeCluster()`.

`makeClusterPSOCK` *Create a PSOCK cluster of* R *workers for parallel processing*

Description

The `makeClusterPSOCK()` function creates a cluster of R workers for parallel processing. These R workers may be background R sessions on the current machine, R sessions on external machines (local or remote), or a mix of such. For external workers, the default is to use SSH to connect to those external machines. This function works similarly to `makePSOCKcluster()` of the **parallel** package, but provides additional and more flexibility options for controlling the setup of the system calls that launch the background R workers, and how to connect to external machines.

Usage

```
makeClusterPSOCK(workers, makeNode = makeNodePSOCK, port = c("auto",
  "random"), ..., autoStop = FALSE, verbose = getOption("future.debug",
  FALSE))

makeNodePSOCK(worker = "localhost", master = NULL, port,
  connectTimeout = getOption("future.makeNodePSOCK.connectTimeout",
  as.numeric(Sys.getenv("R_FUTURE_MAKENODEPSOCK_CONNECTTIMEOUT", 2 * 60))),
  timeout = getOption("future.makeNodePSOCK.timeout",
  as.numeric(Sys.getenv("R_FUTURE_MAKENODEPSOCK_TIMEOUT", 30 * 24 * 60 *
  60))), rscript = NULL, homogeneous = NULL, rscript_args = NULL,
  rscript_startup = NULL, rscript_libs = NULL, methods = TRUE,
  useXDR = TRUE, outfile = "/dev/null", renice = NA_integer_,
  rshcmd = getOption("future.makeNodePSOCK.rshcmd",
  Sys.getenv("R_FUTURE_MAKENODEPSOCK_RSHCMD")), user = NULL,
  revtunnel = TRUE, rshlogfile = NULL,
  rshopts = getOption("future.makeNodePSOCK.rshopts",
  Sys.getenv("R_FUTURE_MAKENODEPSOCK_RSHOPTS")), rank = 1L,
  manual = FALSE, dryrun = FALSE, verbose = FALSE)
```

Arguments

`workers`	The hostnames of workers (as a character vector) or the number of localhost workers (as a positive integer).
`makeNode`	A function that creates a `"SOCKnode"` or `"SOCK0node"` object, which represents a connection to a worker.

port — The port number of the master used for communicating with all the workers (via socket connections). If an integer vector of ports, then a random one among those is chosen. If "random", then a random port in 11000:11999 is chosen. If "auto" (default), then the default is taken from environment variable R_PARALLEL_PORT, otherwise "random" is used. *Note, do not use this argument to specify the port number used by* rshcmd*, which typically is an SSH client. Instead, if the SSH daemon runs on a different port than the default 22, specify the SSH port by appending it to the hostname, e.g.* "remote.server.org:2200" *or via SSH options* -p*, e.g.* rshopts = c("-p", "2200").

... — Optional arguments passed to makeNode(workers[i], ..., rank = i) where i = seq_along(workers).

autoStop — If TRUE, the cluster will be automatically stopped

verbose — If TRUE, informative messages are outputted.

worker — The hostname or IP number of the machine where the worker should run.

master — The hostname or IP number of the master / calling machine, as known to the workers. If NULL (default), then the default is Sys.info()[["nodename"]] unless worker is *localhost* or revtunnel = TRUE in case it is "localhost".

connectTimeout — The maximum time (in seconds) allowed for each socket connection between the master and a worker to be established (defaults to 2 minutes). *See note below on current lack of support on Linux and macOS systems.*

timeout — The maximum time (in seconds) allowed to pass without the master and a worker communicate with each other (defaults to 30 days).

rscript, homogeneous — The system command for launching Rscript on the worker and whether it is installed in the same path as the calling machine or not. For more details, see below.

rscript_args — Additional arguments to Rscript (as a character vector). This argument can be used to customize the R environment of the workers before they launches. For instance, use rscript_args = c("-e", shQuote('setwd("/path/to")')) to set the working directory to '/path/to' on *all* workers.

rscript_startup — An R expression or a character vector of R code, or a list with a mix of these, that will be evaluated on the R worker prior to launching the worker's event loop. For instance, use rscript_startup = 'setwd("/path/to")' to set the working directory to '/path/to' on *all* workers.

rscript_libs — A character vector of R library paths that will be used for the library search path of the R workers. An asterisk ("*") will be resolved as the current .libPaths() on the worker. That is, to prepend a folder, instead of replacing the existing ones, use rscript_libs = c("new_folder", "*").

methods — If TRUE, then the **methods** package is also loaded.

useXDR — If TRUE, the communication between master and workers, which is binary, will use big-endian (XDR).

outfile — Where to direct the stdout and stderr connection output from the workers. If NULL, then no redirection of output is done, which means that the output is relayed in the terminal on the local computer. On Windows, the output is only relayed when running R from a terminal but not from a GUI.

renice	A numerical 'niceness' (priority) to set for the worker processes.
rshcmd, rshopts	The command (character vector) to be run on the master to launch a process on another host and any additional arguments (character vector). These arguments are only applied if machine is not *localhost*. For more details, see below.
user	(optional) The user name to be used when communicating with another host.
revtunnel	If TRUE, a reverse SSH tunnel is set up for each worker such#' that the worker R process sets up a socket connection to its local port (port - rank + 1) which then reaches the master on port port. If FALSE, then the worker will try to connect directly to port port on master. For more details, see below.
rshlogfile	(optional) If a filename, the output produced by the rshcmd call is logged to this file, of if TRUE, then it is logged to a temporary file. The log file name is available as an attribute as part of the return node object. *Warning: This only works with SSH clients that support option* -E out.log.
rank	A unique one-based index for each worker (automatically set).
manual	If TRUE the workers will need to be run manually. The command to run will be displayed.
dryrun	If TRUE, nothing is set up, but a message suggesting how to launch the worker from the terminal is outputted. This is useful for troubleshooting.

Value

An object of class c("SOCKcluster", "cluster") consisting of a list of "SOCKnode" or "SOCK0node" workers.

makeNodePSOCK() returns a "SOCKnode" or "SOCK0node" object representing an established connection to a worker.

Definition of *localhost*

A hostname is considered to be *localhost* if it equals:

- "localhost",
- "127.0.0.1", or
- Sys.info()[["nodename"]].

It is also considered *localhost* if it appears on the same line as the value of Sys.info()[["nodename"]] in file '/etc/hosts'.

Default SSH client and options (arguments rshcmd and rshopts)

Arguments rshcmd and rshopts are only used when connecting to an external host.

The default method for connecting to an external host is via SSH and the system executable for this is given by argument rshcmd. The default is given by option future.makeNodePSOCK.rshcmd. If that is not set, then the default is to use ssh. Most Unix-like systems, including macOS, have ssh preinstalled on the PATH. This is also true for recent Windows 10 (since version 1803; April 2018) (*).

For *Windows systems prior to Windows 10*, it is less common to find `ssh` on the `PATH`. Instead it is more likely that such systems have the `PuTTY` software and its SSH client `plink` installed. PuTTY puts itself on the system `PATH` when installed, meaning this function will find PuTTY automatically if installed. If not, to manually set specify PuTTY as the SSH client, specify the absolute pathname of 'plink.exe' in the first element and option `-ssh` in the second as in `rshcmd = c("C:/Path/PuTTY/plink.exe", "-ssh"`. This is because all elements of `rshcmd` are individually "shell" quoted and element `rshcmd[1]` must be on the system `PATH`.

Furthermore, when running R from RStudio on Windows, the `ssh` client that is distributed with RStudio will be also be considered. This client, which is from MinGW MSYS, is search for in the folder given by the `RSTUDIO_MSYS_SSH` environment variable - a variable that is (only) set when running RStudio.

You can override the default set of SSH clients that are searched for by specifying them in `rshcmd` using the format `<...>`, e.g. `rshcmd = c("<rstudio-ssh>", "<putty-plink>", "<ssh>")`. See below for examples.

If no SSH-client is found, an informative error message is produced.

(*) *Known issue with the Windows 10 SSH client: There is a bug in the SSH client of Windows 10 that prevents it to work with reverse SSH tunneling (*https://github.com/PowerShell/Win32-OpenSSH/issues/1265*; Oct 2018). Because of this, it is recommended to use the PuTTY SSH client or the RStudio SSH client until this bug has been resolved in Windows 10.*

Additional SSH options may be specified via argument `rshopts`, which defaults to option `future.makeNodePSOCK.rshopts`. For instance, a private SSH key can be provided as `rshopts = c("-i", "~/.ssh/my_private_key")`. PuTTY users should specify a PuTTY PPK file, e.g. `rshopts = c("-i", "C:/Users/joe/.ssh/my_keys.ppk")`. Contrary to `rshcmd`, elements of `rshopts` are not quoted.

Accessing external machines that prompts for a password

IMPORTANT: With one exception, it is not possible to for these functions to log in and launch R *workers on external machines that requires a password to be entered manually for authentication.* The only known exception is the PuTTY client on Windows for which one can pass the password via command-line option `-pw`, e.g. `rshopts = c("-pw", "MySecretPassword")`.

Note, depending on whether you run R in a terminal or via a GUI, you might not even see the password prompt. It is also likely that you cannot enter a password, because the connection is set up via a background system call.

The poor man's workaround for setup that requires a password is to manually log into the each of the external machines and launch the R workers by hand. For this approach, use `manual = TRUE` and follow the instructions which include cut'n'pasteable commands on how to launch the worker from the external machine.

However, a much more convenient and less tedious method is to set up key-based SSH authentication between your local machine and the external machine(s), as explain below.

Accessing external machines via key-based SSH authentication

The best approach to automatically launch R workers on external machines over SSH is to set up key-based SSH authentication. This will allow you to log into the external machine without have to enter a password.

Key-based SSH authentication is taken care of by the SSH client and not R. To configure this, see the manuals of your SSH client or search the web for "ssh key authentication".

Reverse SSH tunneling

The default is to use reverse SSH tunneling (`revtunnel = TRUE`) for workers running on other machines. This avoids the complication of otherwise having to configure port forwarding in firewalls, which often requires static IP address as well as privileges to edit the firewall, something most users don't have. It also has the advantage of not having to know the internal and / or the public IP address / hostname of the master. Yet another advantage is that there will be no need for a DNS lookup by the worker machines to the master, which may not be configured or is disabled on some systems, e.g. compute clusters.

Default value of argument `rscript`

If `homogeneous` is FALSE, the `rscript` defaults to `"Rscript"`, i.e. it is assumed that the `Rscript` executable is available on the `PATH` of the worker. If `homogeneous` is TRUE, the `rscript` defaults to `file.path(R.home("bin"), "Rscript")`, i.e. it is basically assumed that the worker and the caller share the same file system and R installation.

Default value of argument `homogeneous`

The default value of `homogeneous` is TRUE if and only if either of the following is fulfilled:

- `worker` is *localhost*
- `revtunnel` is FALSE and `master` is *localhost*
- `worker` is neither an IP number nor a fully qualified domain name (FQDN). A hostname is considered to be a FQDN if it contains one or more periods

In all other cases, `homogeneous` defaults to FALSE.

Connection time out

Argument `connectTimeout` does *not* work properly on Unix and macOS due to limitation in R itself. For more details on this, please see R-devel thread 'BUG?: On Linux setTimeLimit() fails to propagate timeout error when it occurs (works on Windows)' on 2016-10-26 (https://stat.ethz.ch/pipermail/r-devel/2016-October/073309.html). When used, the timeout will eventually trigger an error, but it won't happen until the socket connection timeout `timeout` itself happens.

Communication time out

If there is no communication between the master and a worker within the `timeout` limit, then the corresponding socket connection will be closed automatically. This will eventually result in an error in code trying to access the connection.

Examples

```
## NOTE: Drop 'dryrun = TRUE' below in order to actually connect.  Add
## 'verbose = TRUE' if you run into problems and need to troubleshoot.


## EXAMPLE: Two workers on the local machine
workers <- c("localhost", "localhost")
```

```
cl <- makeClusterPSOCK(workers, dryrun = TRUE)

## EXAMPLE: Three remote workers
## Setup of three R workers on two remote machines are set up
workers <- c("n1.remote.org", "n2.remote.org", "n1.remote.org")
cl <- makeClusterPSOCK(workers, dryrun = TRUE)


## EXAMPLE: Local and remote workers
## Same setup when the two machines are on the local network and
## have identical software setups
cl <- makeClusterPSOCK(
  workers,
  revtunnel = FALSE, homogeneous = TRUE,
  dryrun = TRUE
)

## EXAMPLE: Remote workers with specific setup
## Setup of remote worker with more detailed control on
## authentication and reverse SSH tunnelling
cl <- makeClusterPSOCK(
  "remote.server.org", user = "johnny",
  ## Manual configuration of reverse SSH tunnelling
  revtunnel = FALSE,
  rshopts = c("-v", "-R 11000:gateway:11942"),
  master = "gateway", port = 11942,
  ## Run Rscript nicely and skip any startup scripts
  rscript = c("nice", "/path/to/Rscript"),
  rscript_args = c("--vanilla"),
  dryrun = TRUE
)

## EXAMPLE: Two workers running in Docker on the local machine
## Setup of 2 Docker workers running rocker/r-parallel
cl <- makeClusterPSOCK(
  rep("localhost", times = 2L),
  ## Launch Rscript inside Docker container
  rscript = c(
    "docker", "run", "--net=host", "rocker/r-parallel",
    "Rscript"
  ),
  ## IMPORTANT: Because Docker runs inside a virtual machine (VM) on macOS
  ## and Windows (not Linux), when the R worker tries to connect back to
  ## the default 'localhost' it will fail, because the main R session is
  ## not running in the VM, but outside on the host.  To reach the host on
  ## macOS and Windows, make sure to use master = "host.docker.internal"
  # master = "host.docker.internal",  # <= macOS & Windows
  dryrun = TRUE
)


## EXAMPLE: Two workers running in Singularity on the local machine
## Setup of 2 Singularity workers running rocker/r-parallel
```

```
cl <- makeClusterPSOCK(
  rep("localhost", times = 2L),
  ## Launch Rscript inside Docker container
  rscript = c(
    "singularity", "exec", "docker://rocker/r-parallel",
    "Rscript"
  ),
  dryrun = TRUE
)


## EXAMPLE: One worker running in udocker on the local machine
## Setup of a single udocker.py worker running rocker/r-parallel
cl <- makeClusterPSOCK(
  "localhost",
  ## Launch Rscript inside Docker container (using udocker)
  rscript = c(
    "udocker.py", "run", "rocker/r-parallel",
    "Rscript"
  ),
  ## Manually launch parallel workers
  ## (need double shQuote():s because udocker.py drops one level)
  rscript_args = c(
    "-e", shQuote(shQuote("parallel:::.slaveRSOCK()"))
  ),
  dryrun = TRUE
)


## EXAMPLE: Remote worker running on AWS
## Launching worker on Amazon AWS EC2 running one of the
## Amazon Machine Images (AMI) provided by RStudio
## (http://www.louisaslett.com/RStudio_AMI/)
public_ip <- "1.2.3.4"
ssh_private_key_file <- "~/.ssh/my-private-aws-key.pem"
cl <- makeClusterPSOCK(
  ## Public IP number of EC2 instance
  public_ip,
  ## User name (always 'ubuntu')
  user = "ubuntu",
  ## Use private SSH key registered with AWS
  rshopts = c(
    "-o", "StrictHostKeyChecking=no",
    "-o", "IdentitiesOnly=yes",
    "-i", ssh_private_key_file
  ),
  ## Set up .libPaths() for the 'ubuntu' user
  ## and then install the future package
  rscript_startup = quote(local({
    p <- Sys.getenv("R_LIBS_USER")
    dir.create(p, recursive = TRUE, showWarnings = FALSE)
    .libPaths(p)
    install.packages("future")
```

```
  })),
  dryrun = TRUE
)


## EXAMPLE: Remote worker running on GCE
## Launching worker on Google Cloud Engine (GCE) running a
## container based VM (with a #cloud-config specification)
public_ip <- "1.2.3.4"
user <- "johnny"
ssh_private_key_file <- "~/.ssh/google_compute_engine"
cl <- makeClusterPSOCK(
  ## Public IP number of GCE instance
  public_ip,
  ## User name (== SSH key label (sic!))
  user = user,
  ## Use private SSH key registered with GCE
  rshopts = c(
    "-o", "StrictHostKeyChecking=no",
    "-o", "IdentitiesOnly=yes",
    "-i", ssh_private_key_file
  ),
  ## Launch Rscript inside Docker container
  rscript = c(
    "docker", "run", "--net=host", "rocker/r-parallel",
    "Rscript"
  ),
  dryrun = TRUE
)


## EXAMPLE: Remote worker running on Linux from Windows machine
## Connect to remote Unix machine 'remote.server.org' on port 2200
## as user 'bob' from a Windows machine with PuTTY installed.
## Using the explicit special rshcmd = "<putty-plink>", will force
## makeClusterPSOCK() to search for and use the PuTTY plink software,
## preventing it from using other SSH clients on the system search PATH.
cl <- makeClusterPSOCK(
  "remote.server.org", user = "bob",
  rshcmd = "<putty-plink>",
  rshopts = c("-P", 2200, "-i", "C:/Users/bobby/.ssh/putty.ppk"),
  dryrun = TRUE
)


## EXAMPLE: Remote worker running on Linux from RStudio on Windows
## Connect to remote Unix machine 'remote.server.org' on port 2200
## as user 'bob' from a Windows machine via RStudio's SSH client.
## Using the explicit special rshcmd = "<rstudio-ssh>", will force
## makeClusterPSOCK() to use the SSH client that comes with RStudio,
## preventing it from using other SSH clients on the system search PATH.
cl <- makeClusterPSOCK(
  "remote.server.org", user = "bob", rshcmd = "<rstudio-ssh>",
```

```
  dryrun = TRUE
)
```

`multicore` *Create a multicore future whose value will be resolved asynchronously in a forked parallel process*

Description

A multicore future is a future that uses multicore evaluation, which means that its *value is computed and resolved in parallel in another process*.

Usage

```
multicore(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE,
  workers = availableCores(constraints = "multicore"),
  earlySignal = FALSE, label = NULL, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment from where global objects should be identified.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	If `FALSE` (default), the future is resolved eagerly (starting immediately), otherwise not.
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	(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()`.
`workers`	A positive numeric scalar or a function specifying the maximum number of parallel futures that can be active at the same time before blocking. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return a numeric scalar.
`earlySignal`	Specified whether conditions should be signaled as soon as possible or not.
`label`	An optional character string label attached to the future.
`...`	Additional named elements passed to `Future()`.

Details

This function will block if all cores are occupied and will be unblocked as soon as one of the already running multicore futures is resolved. For the total number of cores available including the current/main R process, see `availableCores()`.

Not all operating systems support process forking and thereby not multicore futures. For instance, forking is not supported on Microsoft Windows. Moreover, process forking may break some R environments such as RStudio. Because of this, the future package disables process forking also in such cases. See `supportsMulticore()` for details. Trying to create multicore futures on non-supported systems or when forking is disabled will result in multicore futures falling back to becoming sequential futures.

The preferred way to create an multicore future is not to call this function directly, but to register it via `plan(multicore)` such that it becomes the default mechanism for all futures. After this `future()` and `%<-%` will create *multicore futures*.

Value

A MulticoreFuture If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using an sequential future. This is also the case whenever multicore processing is not supported, e.g. on Windows.

See Also

For processing in multiple background R sessions, see multisession futures. For multicore processing with fallback to multisession where the former is not supported, see multiprocess futures.

Use `availableCores()` to see the total number of cores that are available for the current R session. Use `availableCores("multicore") > 1L` to check whether multicore futures are supported or not on the current system.

Examples

```
## Use multicore futures
plan(multicore)

## A global variable
a <- 0

## Create multicore future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A multicore future is evaluated in a separate forked
## process.  Changing the value of a global variable
## will not affect the result of the future.
a <- 7
print(a)
```

```
v <- value(f)
print(v)
stopifnot(v == 0)
```

`multiprocess` *Create a multiprocess future whose value will be resolved asynchronously using multicore or a multisession evaluation*

Description

A multiprocess future is a future that uses multicore evaluation if supported, otherwise it uses multisession evaluation. Regardless, its *value is computed and resolved in parallel in another process*.

Usage

```
multiprocess(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE,
  workers = availableCores(), gc = FALSE, earlySignal = FALSE,
  label = NULL, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment from where global objects should be identified.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	If `FALSE` (default), the future is resolved eagerly (starting immediately), otherwise not.
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	(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()`.
`workers`	A positive numeric scalar or a function specifying the maximum number of parallel futures that can be active at the same time before blocking. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return a numeric scalar.
`gc`	If TRUE, the garbage collector run (in the process that evaluated the future) only after the value of the future is collected. Exactly when the values are collected may depend on various factors such as number of free workers and whether `earlySignal` is TRUE (more frequently) or FALSE (less frequently). *Some types of futures ignore this argument.*
`earlySignal`	Specified whether conditions should be signaled as soon as possible or not.
`label`	An optional character string label attached to the future.
`...`	Additional named elements passed to `Future()`.

Value

A MultiprocessFuture implemented as either a MulticoreFuture or a MultisessionFuture.

See Also

Internally `multicore()` and `multisession()` are used.

Examples

```
## Use multiprocess futures
plan(multiprocess)

## A global variable
a <- 0

## Create multicore future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A multiprocess future is evaluated in a separate R process.
## Changing the value of a global variable will not affect
## the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)
```

`multisession` *Create a multisession future whose value will be resolved asynchronously in a parallel* R *session*

Description

A multisession future is a future that uses multisession evaluation, which means that its *value is computed and resolved in parallel in another* R *session*.

Usage

```
multisession(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE, persistent = FALSE,
  workers = availableCores(), gc = FALSE, earlySignal = FALSE,
  label = NULL, ...)
```

Arguments

expr	An R expression.
envir	The environment from where global objects should be identified.
substitute	If TRUE, argument expr is substitute():ed, otherwise not.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
seed	(optional) A L'Ecuyer-CMRG RNG seed.
globals	(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().
persistent	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
workers	A positive numeric scalar or a function specifying the maximum number of parallel futures that can be active at the same time before blocking. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return a numeric scalar.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) only after the value of the future is collected. Exactly when the values are collected may depend on various factors such as number of free workers and whether earlySignal is TRUE (more frequently) or FALSE (less frequently). *Some types of futures ignore this argument.*
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
label	An optional character string label attached to the future.
...	Additional named elements passed to Future().

Details

The background R sessions (the "workers") are created using makeClusterPSOCK().

The multisession() function will block if all available R session are occupied and will be unblocked as soon as one of the already running multisession futures is resolved. For the total number of R sessions available including the current/main R process, see availableCores().

A multisession future is a special type of cluster future.

The preferred way to create an multisession future is not to call this function directly, but to register it via plan(multisession) such that it becomes the default mechanism for all futures. After this future() and %<-% will create *multisession futures*.

Value

A MultisessionFuture. If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using a lazy future.

See Also

For processing in multiple forked R sessions, see multicore futures. For multicore processing with fallback to multisession where the former is not supported, see multiprocess futures.

Use `availableCores()` to see the total number of cores that are available for the current R session.

Examples

```
## Use multisession futures
plan(multisession)

## A global variable
a <- 0

## Create multicore future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A multisession future is evaluated in a separate R session.
## Changing the value of a global variable will not affect
## the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)
```

---

nbrOfWorkers *Get the number of workers available*

Description

Get the number of workers available

Usage

```
nbrOfWorkers(evaluator = NULL)
```

Arguments

evaluator	A future evaluator function. If NULL (default), the current evaluator as returned by plan() is used.

Value

A number in [1, Inf].

Examples

```
plan(multiprocess)
nbrOfWorkers()  ## == availableCores()

plan(sequential)
nbrOfWorkers()  ## == 1
```

---

plan *Plan how to resolve a future*

Description

This function allows *the user* to plan the future, more specifically, it specifies how future():s are resolved, e.g. sequentially or in parallel.

Usage

```
plan(strategy = NULL, ..., substitute = TRUE, .skip = FALSE,
  .call = TRUE, .cleanup = TRUE, .init = TRUE)
```

Arguments

strategy	The evaluation function (or name of it) to use for resolving a future. If NULL, then the current strategy is returned.
...	Additional arguments overriding the default arguments of the evaluation function. Which additional arguments are supported depends on what evaluation function is used, e.g. several support argument workers but not all. For details, see the individual functions of which some are linked to below.
substitute	If TRUE, the strategy expression is substitute():d, otherwise not.
.skip	(internal) If TRUE, then attempts to set a strategy that is the same as what is currently in use, will skipped.
.call	(internal) Used for recording the call to this function.
.cleanup	(internal) Used to stop implicitly started clusters.
.init	(internal) Used to initiate workers.

Details

The default strategy is `sequential`, but the default can be configured by option '`future.plan`' and, if that is not set, system environment variable `R_FUTURE_PLAN`. To reset the strategy back to the default, use `plan("default")`.

Value

If a new strategy is chosen, then the previous one is returned (invisible), otherwise the current one is returned (visibly).

Implemented evaluation strategies

- `sequential`: Resolves futures sequentially in the current R process.
- `transparent`: Resolves futures sequentially in the current R process and assignments will be done to the calling environment. Early stopping is enabled by default.
- `multisession`: Resolves futures asynchronously (in parallel) in separate R sessions running in the background on the same machine.
- `multicore`: Resolves futures asynchronously (in parallel) in separate *forked* R processes running in the background on the same machine. Not supported on Windows.
- `multiprocess`: If multicore evaluation is supported, that will be used, otherwise multisession evaluation will be used.
- `cluster`: Resolves futures asynchronously (in parallel) in separate R sessions running typically on one or more machines.
- `remote`: Resolves futures asynchronously in a separate R session running on a separate machine, typically on a different network.

Other package may provide additional evaluation strategies. Notably, the **future.batchtools** package implements a type of futures that will be resolved via job schedulers that are typically available on high-performance compute (HPC) clusters, e.g. LSF, Slurm, TORQUE/PBS, Sun Grid Engine, and OpenLava.

To "close" any background workers (e.g. `multisession`), change the plan to something different; `plan(sequential)` is recommended for this.

For package developers

Please refrain from modifying the future strategy inside your packages / functions, i.e. do not call `plan()` 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 future framework - as a developer you can never know what future backends the user have access to. Moreover, by not making any assumptions about what backends are available, your code will also work automatically with any new backends developed after you wrote your code.

If you think it is necessary to modify the future strategy within a function, then make sure to undo the changes when exiting the function. This can be done using:

```
oplan <- plan()
on.exit(plan(oplan), add = TRUE)
[...]
```

Using plan() in scripts and vignettes

When writing scripts or vignettes that uses futures, try to place any call to `plan()` as far up (as early on) in the code as possible. This will help users to quickly identify where the future plan is set up and allow them to modify it to their computational resources. Even better is to leave it to the user to set the `plan()` prior to `source()`:ing the script or running the vignette. If a '`.future.R`' exists in the current directory and / or in the user's home directory, it is sourced when the **future** package is *loaded*. Because of this, the '`.future.R`' file provides a convenient place for users to set the `plan()`.

Examples

```
a <- b <- c <- NA_real_

# An sequential future
plan(sequential)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs


# A sequential future with lazy evaluation
plan(sequential)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
}) %lazy% TRUE
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs


# A multicore future (specified as a string)
plan("multicore")
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs
```

```
## Multisession futures gives an error on R CMD check on
## Windows (but not Linux or OS X) for unknown reasons.
## The same code works in package tests.


# A multisession future (specified via a string variable)
strategy <- "future::multisession"
plan(strategy)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs
```

`remote` *Create a remote future whose value will be resolved asynchronously in a remote process*

Description

A remote future is a future that uses remote cluster evaluation, which means that its *value is computed and resolved remotely in another process*.

Usage

```
remote(expr, envir = parent.frame(), substitute = TRUE, lazy = FALSE,
  seed = NULL, globals = TRUE, persistent = TRUE, workers = NULL,
  user = NULL, revtunnel = TRUE, gc = FALSE, earlySignal = FALSE,
  myip = NULL, label = NULL, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment from where global objects should be identified.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	If `FALSE` (default), the future is resolved eagerly (starting immediately), otherwise not.
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	(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()`.

persistent	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
workers	A cluster object, a character vector of host names, a positive numeric scalar, or a function. If a character vector or a numeric scalar, a cluster object is created using makeClusterPSOCK(workers). If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return any of the above types.
user	(optional) The user name to be used when communicating with another host.
revtunnel	If TRUE, reverse SSH tunneling is used for the PSOCK cluster nodes to connect back to the master R process. This avoids the hassle of firewalls, port forwarding and having to know the internal / public IP address of the master R session.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) only after the value of the future is collected. Exactly when the values are collected may depend on various factors such as number of free workers and whether earlySignal is TRUE (more frequently) or FALSE (less frequently). *Some types of futures ignore this argument.*
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
myip	The external IP address of this machine. If NULL, then it is inferred using an online service (default).
label	An optional character string label attached to the future.
...	Additional named elements passed to ClusterFuture().

Value

A ClusterFuture.

'remote' versus 'cluster'

The remote plan is a very similar to the cluster plan, but provides more convenient default argument values when connecting to remote machines. Specifically, remote uses persistent = TRUE by default, and it sets homogeneous, revtunnel, and myip "wisely" depending on the value of workers. See below for example on how remote and cluster are related.

Examples

```
## Not run: \donttest{

## Use a remote machine
plan(remote, workers = "remote.server.org")

## Evaluate expression remotely
host %<-% { Sys.info()["nodename"]] }
host
[1] "remote.server.org"


## The following setups are equivalent:
plan(remote, workers = "localhost")
```

```
plan(cluster, workers = "localhost", persistent = TRUE)
plan(cluster, workers = 1L, persistent = TRUE)
plan(multisession, workers = 1L, persistent = TRUE)

## The following setups are equivalent:
plan(remote, workers = "remote.server.org")
plan(cluster, workers = "remote.server.org", persistent = TRUE, homogeneous = FALSE)

## The following setups are equivalent:
cl <- makeClusterPSOCK("remote.server.org")
plan(remote, workers = cl)
plan(cluster, workers = cl, persistent = TRUE)

}
## End(Not run)
```

`resolve` *Resolve one or more futures synchronously*

Description

This function provides an efficient mechanism for waiting for multiple futures in a container (e.g. list or environment) to be resolved while in the meanwhile retrieving values of already resolved futures.

Usage

```
resolve(x, idxs = NULL, result = FALSE, value = result,
  recursive = 0, sleep = 1, progress = getOption("future.progress",
  FALSE), ...)
```

Arguments

`x`	a list, an environment, or a list environment holding futures that should be resolved. May also be a single Future.
`idxs`	(optional) integer or logical index specifying the subset of elements to check.
`result`	If TRUE, the results are retrieved, otherwise not.
`value`	(DEPRECATED) Use argument 'result' instead.
`recursive`	A non-negative number specifying how deep of a recursion should be done. If TRUE, an infinite recursion is used. If FALSE or zero, no recursion is performed.
`sleep`	Number of seconds to wait before checking if futures have been resolved since last time.
`progress`	(DEPRECATED) If TRUE textual progress summary is outputted. If a function, the it is called as `progress(done, total)` every time a future is resolved.
`...`	Not used

Details

This function is resolves synchronously, i.e. it blocks until `x` and any containing futures are resolved.

Value

Returns `x` (regardless of subsetting or not).

See Also

To resolve a future *variable*, first retrieve its Future object using `futureOf()`, e.g. `resolve(futureOf(x))`.

`resolved` *Check whether a future is resolved or not*

Description

Check whether a future is resolved or not

Usage

```
resolved(x, ...)
```

Arguments

`x`	A Future, a list or an environment (which also includes list environment.
`...`	Not used

Details

This method needs to be implemented by the class that implement the Future API. The implementation must never throw an error, but only return either TRUE or FALSE. It should also be possible to use the method for polling the future until it is resolved (without having to wait infinitely long), e.g. `while (!resolved(future)) Sys.sleep(5)`.

Value

A logical of the same length and dimensions as `x`. Each element is TRUE unless the corresponding element is a non-resolved future in case it is FALSE.

sequential *Create a sequential future whose value will be in the current R session*

Description

A sequential future is a future that is evaluated sequentially in the current R session similarly to how R expressions are evaluated in R. The only difference to R itself is that globals are validated by default just as for all other types of futures in this package.

Usage

```
sequential(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = TRUE, local = TRUE,
  earlySignal = FALSE, label = NULL, ...)

transparent(expr, envir = parent.frame(), substitute = TRUE,
  lazy = FALSE, seed = NULL, globals = FALSE, local = FALSE,
  earlySignal = TRUE, label = NULL, ...)
```

Arguments

expr	An R expression.
envir	The environment from where global objects should be identified.
substitute	If TRUE, argument expr is substitute():ed, otherwise not.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
seed	(optional) A L'Ecuyer-CMRG RNG seed.
globals	(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().
local	If TRUE, the expression is evaluated such that all assignments are done to local temporary environment, otherwise the assignments are done in the calling environment.
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
label	An optional character string label attached to the future.
...	Additional named elements passed to Future().

Details

The preferred way to create a sequential future is not to call these functions directly, but to register them via plan(sequential) such that it becomes the default mechanism for all futures. After this future() and %<-% will create *sequential futures*.

Value

A SequentialFuture.

transparent futures

Transparent futures are sequential futures configured to emulate how R evaluates expressions as far as possible. For instance, errors and warnings are signaled immediately and assignments are done to the calling environment (without `local()` as default for all other types of futures). This makes transparent futures ideal for troubleshooting, especially when there are errors.

Examples

```
## Use sequential futures
plan(sequential)

## A global variable
a <- 0

## Create a sequential future
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## Since 'a' is a global variable in future 'f' which
## is eagerly resolved (default), this global has already
## been resolved / incorporated, and any changes to 'a'
## at this point will _not_ affect the value of 'f'.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)
```

---

tweak *Tweak a future function by adjusting its default arguments*

Description

Tweak a future function by adjusting its default arguments

Usage

```
tweak(strategy, ..., penvir = parent.frame())
```

Arguments

strategy	An existing future function or the name of one.
...	Named arguments to replace the defaults of existing arguments.
penvir	The environment used when searching for a future function by its name.

Value

a future function.

See Also

Use plan() to set a future to become the new default strategy.

---

value.Future *The value of a future*

Description

Gets the value of a future. If the future is unresolved, then the evaluation blocks until the future is resolved.

Usage

```
## S3 method for class 'Future'
value(future, stdout = TRUE, signal = TRUE, ...)
```

Arguments

future	A Future.
stdout	If TRUE, any captured standard output is outputted, otherwise not.
signal	A logical specifying whether (conditions) should signaled or be returned as values.
...	Not used.

Details

This method needs to be implemented by the class that implement the Future API.

Value

An R object of any data type.

values *Get all values in a container*

Description

Gets all values in an environment, a list, or a list environment and returns an object of the same class (and dimensions). All future elements are replaced by their corresponding `value()` values. For all other elements, the existing object is kept.

Usage

```
values(x, ...)
```

Arguments

`x`	An environment, a list, or a list environment.
`...`	Additional arguments passed to `value()` of each future.

Value

An object of same type as `x` and with the same names and/or dimensions, if set.

%conditions% *Control whether standard output should be captured or not*

Description

Control whether standard output should be captured or not

Usage

```
fassignment %conditions% capture
```

Arguments

`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`capture`	If TRUE, the standard output will be captured, otherwise not.

%globals% *Specify globals and packages for a future assignment*

Description

Specify globals and packages for a future assignment

Usage

```
fassignment %globals% globals
fassignment %packages% packages
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
globals	(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	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.

%label% *Specify label for a future assignment*

Description

Specify label for a future assignment

Usage

```
fassignment %label% label
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
label	An optional character string label attached to the future.

%lazy% *Control lazy / eager evaluation for a future assignment*

Description

Control lazy / eager evaluation for a future assignment

Usage

```
fassignment %lazy% lazy
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.

%plan% *Use a specific plan for a future assignment*

Description

Use a specific plan for a future assignment

Usage

```
fassignment %plan% strategy
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
strategy	The mechanism for how the future should be resolved. See plan() for further details.

See Also

The plan() function sets the default plan for all futures.

%seed% *Set random seed for future assignment*

Description

Set random seed for future assignment

Usage

```
fassignment %seed% seed
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
seed	(optional) A L'Ecuyer-CMRG RNG seed.

%stdout% *Control whether standard output should be captured or not*

Description

Control whether standard output should be captured or not

Usage

```
fassignment %stdout% capture
```

Arguments

fassignment	The future assignment, e.g. `x %<-% { expr }`.
capture	If TRUE, the standard output will be captured, otherwise not.

`%tweak%` *Temporarily tweaks the arguments of the current strategy*

Description

Temporarily tweaks the arguments of the current strategy

Usage

```
fassignment %tweak% tweaks
```

Arguments

`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`tweaks`	A named list (or vector) with arguments that should be changed relative to the current strategy.

Index