Package 'future'

February 19, 2017

Version 1.3.0

Title Unified Parallel and Distributed Processing in R for Everyone

Imports digest, globals (>= 0.8.0), listenv (>= 0.6.0), 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, on in parallel a set of local machines, or distributed on a mix of local and remote machines.
Extensions to this package implements additional backends for processing futures via compute cluster schedulers etc.
Because of its unified API, there is no need to modify 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

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

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

RoxygenNote 6.0.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2017-02-19 10:53:35

R topics documented:

`as.cluster` *Coerce an object to a cluster object*

Description

Coerce an object to a cluster object

Usage

```
as.cluster(x, ...)

## S3 method for class 'cluster'
as.cluster(x, ...)

## S3 method for class 'list'
as.cluster(x, ...)
```

```
## S3 method for class 'SOCKnode'
as.cluster(x, ...)

## S3 method for class 'SOCK0node'
as.cluster(x, ...)

## S3 method for class 'cluster'
c(..., recursive = FALSE)
```

Arguments

x	An object to be coerced.
...	Additional arguments passed to the underlying coercion method. For `c(...)`, the clusters and cluster nodes to be combined.
recursive	Not used.

Value

An object of class `cluster`.

`c(...)` combine multiple clusters and / or cluster nodes into one cluster returned as an of class `cluster`.

---

backtrace *Back trace the expressions evaluated before a condition was caught*

Description

Back trace the expressions evaluated before a condition was caught

Usage

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

Arguments

future	The future with a caught condition.
envir	the environment where to locate the future.
...	Not used.

Value

A list of calls.

`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 to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* `plan(lazy)` is used).
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
`persistent`	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
`workers`	A cluster object created by `makeCluster()`.
`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) after the value of the future is collected.
`earlySignal`	Specified whether conditions should be signaled as soon as possible or not.
`label`	An optional character string label attached to the future.
`...`	Additional arguments passed to the "evaluator".

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)
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, lazy = NA,
  seed = NULL, globals = TRUE, evaluator = plan("next"), ...)

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

x %<-% value

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

Arguments

`expr, value`	An R expression to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* `plan(lazy)` is used).
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
`evaluator`	The actual function that evaluates the future expression and returns a Future. 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()`.
`...`	Additional arguments passed to the "evaluator".
`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 needs to be explicitly bracketed, e.g. `x %<-% { a + b }`.

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`.

Until deprecated `plan(lazy)` is defunct, the default (`lazy = NA`) is eager evaluation *unless* `plan(lazy)` is set.

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.

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


## (1) 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)
```

```
## (2) 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)


## (3) 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)


## (4) Explit future assignments (evaluated in parallel)
futureAssign("fitA", lm(y ~ x, weights = w))
futureAssign("fitB", lm(y ~ x - 1, weights = w))
futureAssign("fitC", {
  w <- 1 + abs(x)
  lm(y ~ x, weights = w)
})
print(fitA)
print(fitB)
print(fitC)
```

`Future-class` *A future represents a value that will be available at some point in the future*

Description

A *future* is an abstraction for a *value* that may available at some point in the future. A future can either be `unresolved` or `resolved`, a state which can be checked with `resolved()`. As long as it is *unresolved*, the value is not available. As soon as it is *resolved*, the value is available via `value()`.

Usage

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

Arguments

expr	An R expression.
envir	The environment in which the evaluation is done (or inherits from if local is TRUE).
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.
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.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) after the value of the future is collected. *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 of the future.

Details

A Future object is itself an environment.

Value

An object of class Future.

See Also

One function that creates a Future is future(). It returns a Future that evaluates an R expression in the future. An alternative approach is to use the %<-% infix assignment operator, which creates a future from the right-hand-side (RHS) R expression and assigns its future value to a variable as a *promise*.

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` *Gets all futures in an object*

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.

makeClusterPSOCK *Create a Parallel Socket Cluster*

Description

Create a Parallel Socket Cluster

Usage

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

makeNodePSOCK(worker = "localhost", master = NULL, port,
  connectTimeout = 2 * 60, timeout = 30 * 24 * 60 * 60, rscript = NULL,
  homogeneous = NULL, rscript_args = NULL, methods = TRUE,
  useXDR = TRUE, outfile = "/dev/null", renice = NA_integer_,
  rshcmd = "ssh", user = NULL, revtunnel = TRUE, rshopts = NULL,
  rank = 1L, manual = FALSE, dryrun = FALSE, verbose = FALSE)
```

Arguments

workers	The host names 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 to 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.
...	Optional arguments passed to makeNode(workers[i], ..., rank=i) where i = seq_along{workers}.
verbose	If TRUE, informative messages are outputted.
worker	The host name or IP number of the machine where the worker should run.
master	The host name 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 the localhost ("localhost" or "127.0.0.1") 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. If `NULL` (default), the default is `"Rscript"` unless `homogenenous` is TRUE, which in case it is `file.path(R.home("bin"), "Rscript")`. Argument `homogenenous` defaults to FALSE, unless `master` is the localhost (`"localhost"` or `"127.0.0.1"`).

`rscript_args`
: Additional arguments to `Rscript` (as a character vector).

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

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

`outfile`
: Where to direct the stdout and stderr connection output from the workers.

`renice`
: A numerical 'niceness' (priority) to set for the worker processes.

`rshcmd`
: The command to be run on the master to launch a process on another host. Only applicable if `machine` is not localhost.

`user`
: (optional) The user name to be used when communicating with another host.

`revtunnel`
: If TRUE, a reverse SSH tunneling is set up for each worker such that the worker R process sets up a socket connect 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`.

`rshopts`
: Additional arguments to `rshcmd` (as a character vector).

`rank`
: A unique one-based index for each worker (automatically set).

`manual`
: If TRUE the workers will need to be run manually.

`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.

Details

The `makeClusterPSOCK()` function is similar to `makePSOCKcluster` of the **parallel** package, but provides more flexibility in controlling the setup of the system calls that launch the background R workers and how to connect to external machines.

The default is to use reverse SSH tunneling 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 but which also most users don't have privileges to do themselves. It also has the advantage of not having to know the internal and / or the public IP address / host name of the master.

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.

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.

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 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.

Examples

```
## 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)

## 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
)

## 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
)

## Setup of Docker worker running rocker/r-base
## (requires installation of future package)
cl <- makeClusterPSOCK(
  "localhost",
  ## Launch Rscript inside Docker container
  rscript = c(
    "docker", "run", "--net=host", "rocker/r-base",
    "Rscript"
  ),
  ## Install future package
  rscript_args = c(
    "-e", shQuote("install.packages('future')")
  ),
  dryrun = TRUE
)
```

```
## Setup of udocker.py worker running rocker/r-base
## (requires installation of future package and extra quoting)
cl <- makeClusterPSOCK(
  "localhost",
  ## Launch Rscript inside Docker container (using udocker)
  rscript = c(
    "udocker.py", "run", "rocker/r-base",
    "Rscript"
  ),
  ## Install future package and manually launch parallel workers
  ## (need double shQuote():s because udocker.py drops one level)
  rscript_args = c(
    "-e", shQuote(shQuote("install.packages('future')")),
    "-e", shQuote(shQuote("parallel:::.slaveRSOCK()"))
  ),
  dryrun = TRUE
)


## 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
  ## install future package
  rscript_args = c(
    "-e", shQuote("local({
      p <- Sys.getenv('R_LIBS_USER')
      dir.create(p, recursive = TRUE, showWarnings = FALSE)
      .libPaths(p)
    })"),
    "-e", shQuote("install.packages('future')")
  ),
  dryrun = TRUE
)


## 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-base",
    "Rscript"
  ),
  ## Install future package
  rscript_args = c(
    "-e", shQuote("install.packages('future')")
  ),
  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 to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* `plan(lazy)` is used).

seed	(optional) A L'Ecuyer-CMRG RNG seed.
globals	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
workers	The maximum number of multicore futures that can be active at the same time before blocking.
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
label	An optional character string label attached to the future.
...	Additional arguments passed to the "evaluator".

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 systems support multicore futures. For instance, it is not supported on Microsoft Windows. Trying to create multicore futures on non-supported systems will silently fall back to using sequential futures, which effectively corresponds to a multicore future that can handle one parallel process (the current one) before blocking.

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 to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	If `FALSE` (default), the future is resolved eagerly (immediately), otherwise not.
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	(optional) a logical, a character vector, or a named list for controlling how globals are handled. For details, see section 'Globals used by future expressions' in the help for `future()`.
`workers`	The maximum number of multiprocess futures that can be active at the same time before blocking.
`gc`	If TRUE, the garbage collector run (in the process that evaluated the future) after the value of the future is collected.
`earlySignal`	Specified whether conditions should be signaled as soon as possible or not.
`label`	An optional character string label attached to the future.
`...`	Not used.

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 to be evaluated.
envir	The environment from where global objects should be identified. Depending on the future strategy (the evaluator), it may also be the environment in which the expression is evaluated.
substitute	If TRUE, argument expr is substitute():ed, otherwise not.
lazy	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* plan(lazy) is used).
seed	(optional) A L'Ecuyer-CMRG RNG seed.
globals	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
persistent	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
workers	The maximum number of multisession futures that can be active at the same time before blocking.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) after the value of the future is collected.
earlySignal	Specified whether conditions should be signaled as soon as possible or not.
label	An optional character string label attached to the future.
...	Additional arguments passed to the "evaluator".

Details

This 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.

Known issues

In the current implementation, *all* background R sessions are allocated and launched in the background *as soon as the first multisession future is created*. This means that more R sessions may be running than what will ever be used. The reason for this is that background sessions are currently created using `makeCluster()`, which requires that all R sessions are created at once.

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 *Gets the number of workers available*

Description

Gets 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(multiprocess, workers=2)
nbrOfWorkers()  ## == 2

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

---

plan *Plan how to resolve a future*

Description

This function allows you 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, .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.
substitute	If TRUE, the strategy expression is substitute():d, otherwise not.
.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.BatchJobs** 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.

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
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
plan(multisession)
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 to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* `plan(lazy)` is used).
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
`persistent`	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
`workers`	The maximum number of multiprocess futures that can be active at the same time before blocking.
`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) after the value of the future is collected.
`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 arguments passed to the "evaluator".

Details

Note that remote futures use `persistent=TRUE` by default.

Value

A ClusterFuture.

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"

}
## End(Not run)
```

`resolve` *Wait until all existing futures in an environment are resolved*

Description

The environment is first scanned for futures and then the futures are polled until all are resolved. When a resolved future is detected its value is retrieved (optionally). This provides an efficient mechanism for waiting for a set of futures to be resolved and in the meanwhile retrieving values of already resolved futures.

Usage

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

Arguments

`x`	an environment holding futures.
`idxs`	subset of elements to check.
`value`	If TRUE, the values are retrieved, otherwise not.
`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`	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

Value

Returns x (regardless of subsetting or not).

See Also

futureOf

`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, ...)

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

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

Arguments

`expr`	An R expression to be evaluated.
`envir`	The environment from where global objects should be identified. Depending on the future strategy (the `evaluator`), it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`lazy`	Specifies whether a future should be resolved lazily or eagerly. The default is eager evaluation (except when the *deprecated* `plan(lazy)` is used).
`seed`	(optional) A L'Ecuyer-CMRG RNG seed.
`globals`	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
`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 arguments passed to the "evaluator".

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)
```

---

supportsMulticore *Check whether multicore processing is supported or not*

Description

Multicore futures are only supported on systems supporting multicore processing. R supports this on most systems, except on Microsoft Windows.

Usage

```
supportsMulticore()
```

Value

TRUE if multicore processing is supported, otherwise FALSE.

See Also

To use multicore futures, use `plan(multicore)`.

`tweak` *Tweaks a future function by adjusting its default arguments*

Description

Tweaks 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, signal = TRUE, ...)
```

Arguments

future	A Future.
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 *Gets all values in an object*

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.

`%globals%` *Specify globals for a future assignment*

Description

Specify globals for a future assignment

Usage

```
fassignment %globals% globals
```

Arguments

`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`globals`	(optional) a logical, a character vector, or a named list for controlling how globals are handled. For details, see section 'Globals used by future expressions' in the help for `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 (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.

%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