Package 'parallelly'

January 4, 2021

Version 1.23.0

Title Enhancing the 'parallel' Package

Imports parallel, tools, utils

Description Utility functions that enhance the 'parallel' package and support the built-in parallel backends of the 'future' package. For example, availableCores() gives the number of CPU cores available to your R process as given by the operating system, 'cgroups' and Linux containers, R options, and environment variables, including those set by job schedulers on high-performance compute clusters. If none is set, it will fall back to parallel::detectCores(). Another example is makeClusterPSOCK(), which is backward compatible with parallel::makePSOCKcluster() while doing a better job in setting up remote cluster workers without the need for configuring the firewall to do port-forwarding to your local computer.

License LGPL (>= 2.1)

LazyLoad TRUE

ByteCompile TRUE

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

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

RoxygenNote 7.1.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2021-01-04 21:20:05 UTC

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`.

Examples

```
cl1 <- makeClusterPSOCK(2, dryrun = TRUE)
cl2 <- makeClusterPSOCK(c("n1", "server.remote.org"), dryrun = TRUE)
cl <- c(cl1, cl2)
print(cl)
```

---

autoStopCluster *Automatically Stop a Cluster when Garbage Collected*

Description

Registers a finalizer to a cluster such that the cluster will be stopped when garbage collected

Usage

```
autoStopCluster(cl, debug = FALSE)
```

Arguments

cl	A cluster object created by for instance makeClusterPSOCK() or parallel::makeCluster().
debug	If TRUE, then debug messages are produced when the cluster is garbage collected.

Value

The cluster object with attribute gcMe set.

See Also

The cluster is stopped using stopCluster(cl).

Examples

```
cl <- makeClusterPSOCK(2, dryrun = TRUE)
cl <- autoStopCluster(cl)
print(cl)
rm(list = "cl")
gc()
```

availableConnections *Number of Available and Free Connections*

Description

The number of connections that can be open at the same time in R is *typically* 128, where the first three are occupied by the always open `stdin()`, `stdout()`, and `stderr()` connections, which leaves 125 slots available for other types of connections. Connections are used in many places, e.g. reading and writing to file, downloading URLs, communicating with parallel R processes over a socket connections, and capturing standard output via text connections.

Usage

```
availableConnections()

freeConnections()
```

Value

A non-negative integer, or `+Inf` if the available number of connections is greated than 16384, which is a limit be set via option '`parallelly.availableConnections.tries`'.

How to increase the limit

This limit of 128 connections can only be changed by rebuilding R from source. The limited is hardcoded as a

```
#define NCONNECTIONS 128
```

in '`src/main/connections.c`'.

How the limit is identified

Since the limit *might* changed, for instance in custom R builds or in future releases of R, we do not want to assume that the limit is 128 for all R installation. Unfortunately, it is not possible to query R for what the limit is. Instead, `availableConnections()` infers it from trial-and-error. until it fails. For efficiency, the result is memoized throughout the current R session.

References

1. 'WISH: Increase limit of maximum number of open connections (currently 125+3)', 2016-07-09, https://github.com/HenrikBengtsson/Wishlist-for-R/issues/28

See Also

`base::showConnections()`.

Examples

```
total <- availableConnections()
message("You can have ", total, " connections open in this R installation")
free <- freeConnections()
message("There are ", free, " connections remaining")
```

`availableCores` *Get Number of Available Cores on The Current Machine*

Description

The current/main R session counts as one, meaning the minimum number of cores available is always at least one.

Usage

```
availableCores(
  constraints = NULL,
  methods = getOption2("future.availableCores.methods", c("system", "nproc",
   "mc.cores", "_R_CHECK_LIMIT_CORES_", "PBS", "SGE", "Slurm", "LSF", "fallback",
    "custom")),
  na.rm = TRUE,
  logical = getOption2("future.availableCores.logical", TRUE),
  default = c(current = 1L),
  which = c("min", "max", "all")
)
```

Arguments

`constraints`	An optional character specifying under what constraints ("purposes") we are requesting the values. For instance, on systems where multicore processing is not supported (i.e. Windows), using `constrains = "multicore"` will force a single core to be reported.
`methods`	A character vector specifying how to infer the number of available cores.
`na.rm`	If TRUE, only non-missing settings are considered/returned.
`logical`	Passed to `detectCores(logical = logical)`, which, if supported, returns the number of logical CPUs (TRUE) or physical CPUs/cores (FALSE). This argument is only if argument `methods` includes `"system"`.
`default`	The default number of cores to return if no non-missing settings are available.
`which`	A character specifying which settings to return. If `"min"`, the minimum value is returned. If `"max"`, the maximum value is returned (be careful!) If `"all"`, all values are returned.

Details

The following settings ("methods") for inferring the number of cores are supported:

- `"system"` - Query `detectCores(logical = logical)`.
- `"nproc"` - On Unix, query system command `nproc`.
- `"mc.cores"` - If available, returns the value of option `mc.cores`. Note that '`mc.cores`' is defined as the number of *additional* R processes that can be used in addition to the main R process. This means that with `mc.cores = 0` all calculations should be done in the main R process, i.e. we have exactly one core available for our calculations. The '`mc.cores`' option defaults to environment variable `MC_CORES` (and is set accordingly when the **parallel** package is loaded). The '`mc.cores`' option is used by for instance `mclapply()` of the **parallel** package.
- `"PBS"` - Query TORQUE/PBS environment variables `PBS_NUM_PPN` and `NCPUS`. Depending on PBS system configuration, these *resource* parameter may or may not default to one. An example of a job submission that results in this is qsub -l nodes=1:ppn=2, which requests one node with two cores.
- `"SGE"` - Query Sun/Oracle Grid Engine (SGE) environment variable `NSLOTS`. An example of a job submission that results in this is qsub -pe smp 2 (or qsub -pe by_node 2), which requests two cores on a single machine.
- `"Slurm"` - Query Simple Linux Utility for Resource Management (Slurm) environment variable `SLURM_CPUS_PER_TASK`. This may or may not be set. It can be set when submitting a job, e.g. sbatch --cpus-per-task=2 hello.sh or by adding #SBATCH --cpus-per-task=2 to the '`hello.sh`' script. If `SLURM_CPUS_PER_TASK` is not set, then it will fall back to use `SLURM_CPUS_ON_NODE` if the job is a single-node job (`SLURM_JOB_NUM_NODES` is 1), e.g. sbatch --ntasks=2 hello.sh.
- `"LSF"` - Query Platform Load Sharing Facility (LSF) environment variable `LSB_DJOB_NUMPROC`. Jobs with multiple (CPU) slots can be submitted on LSF using bsub -n 2 -R "span[hosts=1]" < hello.sh.
- `"custom"` - If option '`future.availableCores.custom`' is set and a function, then this function will be called (without arguments) and it's value will be coerced to an integer, which will be interpreted as a number of available cores. If the value is NA, then it will be ignored.

For any other value of a `methods` element, the R option with the same name is queried. If that is not set, the system environment variable is queried. If neither is set, a missing value is returned.

Value

Return a positive (>= 1) integer. If `which = "all"`, then more than one value may be returned. Together with `na.rm = FALSE` missing values may also be returned.

Avoid ending up with zero cores

Note that `availableCores()` may return a single core. Because of this, using something like:

```
ncores <- availableCores() - 1
```

may return zero, which is often not intended. Instead, use:

```
ncores <- max(1, availableCores() - 1)
```

Advanced usage

It is possible to override the maximum number of cores on the machine as reported by `availableCores(methods = "system")`. This can be done by first specifying `options(future.availableCores.methods = "mc.cores")` and then the number of cores to use, e.g. `options(mc.cores = 8)`.

See Also

To get the set of available workers regardless of machine, see `availableWorkers()`.

Examples

```
message(paste("Number of cores available:", availableCores()))

## Not run:
options(mc.cores = 2L)
message(paste("Number of cores available:", availableCores()))

## End(Not run)

## Not run:
## IMPORTANT: availableCores() may return 1L
options(mc.cores = 1L)
ncores <- max(1, availableCores() - 1)
message(paste("Number of cores to use:", ncores))

## End(Not run)

## Not run:
## Use 75% of the cores on the system but never more than four
options(parallelly.availableCores.custom = function() {
  ncores <- max(parallel::detectCores(), 1L, na.rm = TRUE)
  min(0.75 * ncores, 4L)
})
message(paste("Number of cores available:", availableCores()))

## End(Not run)
```

`availableWorkers` *Get Set of Available Workers*

Description

Get Set of Available Workers

Usage

```
availableWorkers(
  methods = getOption2("future.availableWorkers.methods", c("mc.cores",
    "_R_CHECK_LIMIT_CORES_", "PBS", "SGE", "Slurm", "LSF", "custom", "system",
    "fallback")),
  na.rm = TRUE,
  logical = getOption2("future.availableCores.logical", TRUE),
  default = "localhost",
  which = c("auto", "min", "max", "all")
)
```

Arguments

methods	A character vector specifying how to infer the number of available cores.
na.rm	If TRUE, only non-missing settings are considered/returned.
logical	Passed as-is to availableCores().
default	The default set of workers.
which	A character specifying which set / sets to return. If "auto", the first non-empty set found. If "min", the minimum value is returned. If "max", the maximum value is returned (be careful!) If "all", all values are returned.

Details

The default set of workers for each method is rep("localhost",times = availableCores(methods = method,logical = logical)), which means that each will at least use as many parallel workers on the current machine that availableCores() allows for that method.

In addition, the following settings ("methods") are also acknowledged:

- "PBS" - Query TORQUE/PBS environment variable PBS_NODEFILE. If this is set and specifies an existing file, then the set of workers is read from that file, where one worker (node) is given per line. An example of a job submission that results in this is qsub -l nodes = 4:ppn = 2, which requests four nodes each with two cores.
- "SGE" - Query Sun/Oracle Grid Engine (SGE) environment variable PE_HOSTFILE. An example of a job submission that results in this is qsub -pe mpi 8 (or qsub -pe ompi 8), which requests eight cores on a any number of machines.
- "LSF" - Query LSF/OpenLava environment variable LSB_HOSTS.
- "custom" - If option 'future.availableWorkers.custom' is set and a function, then this function will be called (without arguments) and it's value will be coerced to a character vector, which will be interpreted as hostnames of available workers.

Value

Return a character vector of workers, which typically consists of names of machines / compute nodes, but may also be IP numbers.

See Also

To get the number of available workers on the current machine, see availableCores().

Examples

```
message(paste("Available workers:",
        paste(sQuote(availableWorkers()), collapse = ", ")))

## Not run:
options(mc.cores = 2L)
message(paste("Available workers:",
        paste(sQuote(availableWorkers()), collapse = ", ")))

## End(Not run)

## Not run:
## Always use two workers on host 'n1' and one on host 'n2'
options(parallelly.availableWorkers.custom = function() {
  c("n1", "n1", "n2")
})
message(paste("Available workers:",
        paste(sQuote(availableWorkers()), collapse = ", ")))

## End(Not run)
```

`isConnectionValid` *Checks if a Connection is Valid*

Description

Get a unique identifier for an R connection and check whether or not the connection is still valid.

Usage

```
isConnectionValid(con)

connectionId(con)
```

Arguments

`con`	A connection.

Value

`isConnectionValid()` returns TRUE if the connection is still valid, otherwise FALSE. If FALSE, then character attribute `reason` provides an explanation why the connection is not valid.

`connectionId()` returns an non-negative integer, -1, or `NA_integer_`. For connections stdin, stdout, and stderr, 0, 1, and 2, are returned, respectively. For all other connections, an integer greater or equal to 3 based on the connection's internal pointer is returned. A connection that has been serialized, which is no longer valid, has identifier -1. Attribute `raw_id` returns the pointer string from which the above is inferred.

Connection Index versus Connection Identifier

R represents connections as indices using plain integers, e.g. `idx <-as.integer(con)`. The three connections standard input ("stdin"), standard output ("stdout"), and standard error ("stderr") always exists and have indices 0, 1, and 2. Any connection opened beyond these will get index three or greater, depending on availability as given by `base::showConnections()`. To get the connection with a given index, use `base::getConnection()`. **Unfortunately, this index representation of connections is non-robust**, e.g. there are cases where two or more 'connection' objects can end up with the same index and if used, the written output may end up at the wrong destination and files and database might get corrupted. This can for instance happen if `base::closeAllConnections()` is used (*). **In contrast,** `id <- connectionId(con)` **gives an identifier that is unique to that 'connection' object.** This identifier is based on the internal pointer address of the object. The risk for two connections in the same R session to end up with the same pointer address is very small. Thus, in case we ended up in a situation where two connections `con1` and `con2` share the same index - `as.integer(con1) == as.integer(con2)` - they will never share the same identifier - `connectionId(con1) != connectionId(con2)`. Here, `isConnectionValid()` can be used to check which one of these connections, if any, are valid.

(*) Note that there is no good reason for calling `closeAllConnections()` If called, there is a great risk that the files get corrupted etc. See (1) for examples and details on this problem. If you think there is a need to use it, it is much safer to restart R because that is guaranteed to give you a working R session with non-clashing connections. It might also be that `closeAllConnections()` is used because `base::sys.save.image()` is called, which might happen if R is being forced to terminate.

Connections Cannot be Serialized Or Saved

A 'connection' cannot be serialized, e.g. it cannot be saved to file to be read and used in another R session. If attempted, the connection will not be valid. This is a problem that may occur in parallel processing when passing an R object to parallel worker for further processing, e.g. the exported object may hold an internal database connection which will no longer be valid on the worker. When a connection is serialized, its internal pointer address will be invalidated (set to nil). In such cases, `connectionId(con)` returns -1 and `isConnectionValid(con)` returns FALSE.

References

1. 'BUG: A connection object may become corrupt and re-referenced to another connection (PATCH)', 2018-10-30.
2. R-devel thread PATCH: Asserting that 'connection' used has not changed + R_GetConnection2(), 2018-10-31.

See Also

See `base::showConnections()` for currently open connections and their indices. To get a connection by its index, use `base::getConnection()`.

Examples

```
## R represents connections as plain indices
as.integer(stdin())         ## int 0
as.integer(stdout())        ## int 1
```

```
as.integer(stderr())        ## int 2

## The first three connections always exist and are always valid
isConnectionValid(stdin())   ## TRUE
connectionId(stdin())        ## 0L
isConnectionValid(stdout())  ## TRUE
connectionId(stdout())       ## 1L
isConnectionValid(stderr())  ## TRUE
connectionId(stderr())       ## 2L

## Connections cannot be serialized
con <- file(tempfile(), open = "w")
x <- list(value = 42, stderr = stderr(), con = con)
y <- unserialize(serialize(x, connection = NULL))
isConnectionValid(y$stderr)  ## TRUE
connectionId(y$stderr)       ##  2L
isConnectionValid(y$con)     ## FALSE with attribute 'reason'
connectionId(y$con)          ## -1L
close(con)
```

`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 = getOptionOrEnvVar("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 using `stopCluster()` when it is garbage collected, unless already stopped. See also `autoStopCluster()`.
`verbose`	If TRUE, informative messages are outputted.

Details

Creating MPI clusters requires that the **Rmpi** *and* **snow** *packages are installed.*

Value

An object of class `c("RichMPIcluster","MPIcluster","cluster")` 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()`.

Examples

```
## Not run:
if (requireNamespace("Rmpi") && requireNamespace("snow")) {
  cl <- makeClusterMPI(2, autoStop = TRUE)
  print(cl)
  y <- parLapply(cl, X = 1:3, fun = sqrt)
  print(y)
  rm(list = "cl")
}

## End(Not run)
```

`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,
  tries = getOptionOrEnvVar("future.makeNodePSOCK.tries", 3L),
  delay = getOptionOrEnvVar("future.makeNodePSOCK.tries.delay", 15),
  validate = getOptionOrEnvVar("future.makeNodePSOCK.validate", TRUE),
  verbose = getOptionOrEnvVar("future.debug", FALSE)
)

makeNodePSOCK(
  worker = "localhost",
  master = NULL,
  port,
 connectTimeout = getOptionOrEnvVar("future.makeNodePSOCK.connectTimeout", 2 * 60),
 timeout = getOptionOrEnvVar("future.makeNodePSOCK.timeout", 30 * 24 * 60 * 60),
  rscript = NULL,
  homogeneous = NULL,
  rscript_args = NULL,
  rscript_envs = NULL,
  rscript_libs = NULL,
  rscript_startup = NULL,
  methods = TRUE,
  useXDR = getOptionOrEnvVar("future.makeNodePSOCK.useXDR", FALSE),
  outfile = "/dev/null",
  renice = NA_integer_,
  rshcmd = getOptionOrEnvVar("future.makeNodePSOCK.rshcmd", NULL),
  user = NULL,
  revtunnel = TRUE,
  rshlogfile = NULL,
  rshopts = getOptionOrEnvVar("future.makeNodePSOCK.rshopts", NULL),
  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 is chosen from 11000:11999, or from the range specified by environment variable R_FUTURE_RANDOM_PORTS. If "auto" (default), then the default (single) port 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 using stopCluster() when it is garbage collected, unless already stopped. See also autoStopCluster().

`tries, delay` Maximum number of attempts done to launch each node with makeNode() and the delay (in seconds) in-between attempts. If argument port specifies more than one port, e.g. port = "random" then a random port will be drawn and validated at most tries times.

`validate` If TRUE, after the nodes have been created, they are all validated that they work by inquiring about their session information, which is saved in attribute session_info of each node.

`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_envs` A named character vector environment variables to set on worker at startup, e.g. rscript_envs = c(FOO = "3.14","HOME","UNKNOWN"). If an element is not named, then the value of that variable will be used as the name and the value will be the value of Sys.getenv() for that variable. Non-existing environment variables will be dropped. These variables are set using Sys.setenv().

`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 to the default .libPaths()

on the worker. That is, to `prepend` a folder, instead of replacing the existing ones, use `rscript_libs = c("new_folder","*")`. To pass down a non-default library path currently set *on the main* R *session* to the workers, use `rscript_libs = .libPaths()`.

`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.

`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("RichSOCKcluster","SOCKcluster","cluster")` consisting of a list of `"SOCKnode"` or `"SOCK0node"` workers (that also inherit from `RichSOCKnode`).

`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 also be considered. This client, which is from MinGW MSYS, is searched 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). The most recent version that we tested and that did* not *work was OpenSSH_for_Windows_7.7p1, LibreSSL 2.6.5 (*`ssh -V`*) on Windows 10 (version 1909, OS build 18363.720) (*`ver`*). 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.

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.

If specified, argument `rscript` should be a character vector with one more more elements. all elements are automatically shell quoted using `base::shQuote()`, except those that are of format <ENVVAR>=<VALUE>, that is, the ones matching the regular expression '`^[[:alpha:]_][[:alnum:]_]*=.*`". Another exception is when `rscript` inherits from 'AsIs'.

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.

Failing to set up local workers

When setting up a cluster of localhost workers, that is, workers running on the same machine as the master R process, occasionally a connection to a worker ("cluster node") may fail to be set up. When this occurs, an informative error message with troubleshooting suggestions will be produced. The most common reason for such localhost failures is due to port clashes. Retrying will often resolve the problem.

Failing to set up remote workers

A cluster of remote workers runs R processes on external machines. These external R processes are launched over, typically, SSH to the remote machine. For this to work, each of the remote machines needs to have R installed, which preferably is of the same version as what is on the main machine. For this to work, it is required that one can SSH to the remote machines. Ideally, the SSH connections use authentication based on public-private SSH keys such that the set up of the remote workers can be fully automated (see above). If `makeClusterPSOCK()` fails to set up one or more remote R workers, then an informative error message is produced. There are a few reasons for failing to set up remote workers. If this happens, start by asserting that you can SSH to the remote machine and launch '`Rscript`' by calling something like:

```
{local}$ ssh -l alice remote.server.org
{remote}$ Rscript --version
R scripting front-end version 3.6.1 (2019-07-05)
{remote}$ logout
{local}$
```

When you have confirmed the above to work, then confirm that you can achieve the same in a single command-line call;

```
{local}$ ssh -l alice remote.server.org Rscript --version
R scripting front-end version 3.6.1 (2019-07-05)
{local}$
```

The latter will assert that you have proper startup configuration also for *non-interactive* shell sessions on the remote machine.

Another reason for failing to setup remote workers could be that they are running an R version that is not compatible with the version that your main R session is running. For instance, if we run R (>= 3.6.0) locally and the workers run R (< 3.5.0), we will get: Error in unserialize(node$con) : error reading from connection. This is because R (>= 3.6.0) uses serialization format version 3 whereas R (< 3.5.0) only supports version 2. We can see the version of the R workers by adding `rscript_args = c("-e", shQuote("getRversion()"))` when calling `makeClusterPSOCK()`.

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 Linux 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
)
```

`supportsMulticore` *Check If Forked Processing ("multicore") is Supported*

Description

Certain parallelization methods in R rely on *forked* processing, e.g. `parallel::mclapply()`, `parallel::makeCluster(n,type = "FORK")`, `doMC::registerDoMC()`, and `future::plan("multicore")`. Process forking is done by the operating system and support for it in R is restricted to Unix-like operating systems such as Linux, Solaris, and macOS. R running on Microsoft Windows does not support forked processing. In R, forked processing is often referred to as "multicore" processing, which stems from the 'mc' of the `mclapply()` family of functions, which originally was in a package named **multicore** which later was incorporated into the **parallel** package. This function checks whether or not forked (aka "multicore") processing is supported in the current R session.

Usage

```
supportsMulticore(...)
```

Arguments

`...`	Internal usage only.

Value

TRUE if forked processing is supported and not disabled, otherwise FALSE.

Support for process forking

The Microsoft Windows operating system does not support processes forking. Unix-like operating system such as Linux and macOS support forking.

For some R environments it is considered unstable to perform parallel processing based on *forking*. This is for example the case when using RStudio, cf. RStudio Inc. recommends against using forked processing when running R from within the RStudio software. This function detects when running in such an environment and returns `FALSE`, despite the underlying operating system supports forked processing. A warning will also be produced informing the user about this the first time time this function is called in an R session. This warning can be disabled by setting R option '`future.supportsMulticore.unstable`', or environment variable `R_FUTURE_SUPPORTSMULTICORE_UNSTABLE` to `"quiet"`.

Enable or disable forked processing

It is possible to disable forked processing for futures by setting R option '`future.fork.enable`' to `FALSE`. Alternatively, one can set environment variable `R_FUTURE_FORK_ENABLE` to `false`. Analogously, it is possible to override disabled forking by setting one of these to `TRUE`.

Examples

```
## Check whether or not forked processing is supported
supportsMulticore()
```

Index