Package 'parallelly'

January 13, 2023

Version 1.34.0

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://parallelly.futureverse.org, https://github.com/HenrikBengtsson/parallelly

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

RoxygenNote 7.2.3

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2023-01-13 17:30:02 UTC

R topics documented:

as.cluster 2
autoStopCluster 3
availableConnections 4
availableCores 5

availableWorkers 8
freePort 11
isConnectionValid 12
isForkedChild 14
isForkedNode 14
isLocalhostNode 15
isNodeAlive 15
killNode 16
makeClusterMPI 18
makeClusterPSOCK 19
parallelly.options 31
supportsMulticore 34

Index **36**

`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`. A warning will be produced if there are duplicated nodes in the resulting 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.

Details

The cluster is stopped using `stopCluster(cl)`. An alternative to explicitly call this function on an existing `cluster` object, is to create the `cluster` object using `makeClusterPSOCK()` with argument `autoStop = TRUE`.

Value

The cluster object with attribute `gcMe` set.

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 (e.g. `parallel::makeCluster()` and `makeClusterPSOCK()`), and capturing standard output via text connections (e.g. `utils::capture.output()`).

Usage

```
availableConnections()

freeConnections()
```

Value

A non-negative integer, or `+Inf` if the available number of connections is greater 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("parallelly.availableCores.methods", c("system", "cgroups.cpuset",
   "cgroups.cpuquota", "cgroups2.cpu.max", "nproc", "mc.cores", "BiocParallel",
   "_R_CHECK_LIMIT_CORES_", "Bioconductor", "LSF", "PJM", "PBS", "SGE", "Slurm",
    "fallback", "custom")),
  na.rm = TRUE,
  logical = getOption2("parallelly.availableCores.logical", TRUE),
  default = c(current = 1L),
  which = c("min", "max", "all"),
  omit = getOption2("parallelly.availableCores.omit", 0L)
)
```

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 `constraints = "multicore"` will force a single core to be reported. Using `constraints = "connections"`, will append `"connections"` to the `methods` argument. It is possible to specify multiple constraints, e.g. `constraints = c("connections", "multicore")`.
`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). At least as of R 4.2.2, `detectCores()` this argument on Linux. 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" (default), the minimum value is returned. If "max", the maximum value is returned (be careful!) If "all", all values are returned.
omit	(integer; non-negative) Number of cores to not include.

Details

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

- "system" - Query detectCores(logical = logical).
- "cgroups.cpuset" - On Unix, query control group (cgroup) value cpuset.set.
- "cgroups.cpuquota" - On Unix, query control group (cgroup) value cpu.cfs_quota_us / cpu.cfs_period_us.
- "cgroups2.cpu.max" - On Unix, query control group (cgroup v2) values cpu.max.
- "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.
- "connections" - Query the current number of available R connections per freeConnections(). This is the maximum number of socket-based **parallel** cluster nodes that are possible launch, because each one needs its own R connection. The exception is when freeConnections() is zero, then 1L is still returned, because availableCores() should always return a positive integer.
- "BiocParallel" - Query environment variable BIOCPARALLEL_WORKER_NUMBER (integer), which is defined and used by **BiocParallel** (>= 1.27.2). If the former is set, this is the number of cores considered.
- "_R_CHECK_LIMIT_CORES_" - Query environment variable _R_CHECK_LIMIT_CORES_ (logical or "warn") used by R CMD check and set to true by R CMD check --as-cran. If set to a non-false value, then a maximum of 2 cores is considered.
- "Bioconductor" - Query environment variable IS_BIOC_BUILD_MACHINE (logical) used by the Bioconductor (>= 3.16) build and check system. If set to true, then a maximum of 4 cores is considered.
- "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.
- "PJM" - Query Fujitsu Technical Computing Suite (that we choose to shorten as "PJM") environment variables PJM_VNODE_CORE and PJM_PROC_BY_NODE. The first is set when submitted with pjsub -L vnode-core=8 hello.sh.
- "PBS" - Query TORQUE/PBS environment variables PBS_NUM_PPN and NCPUS. Depending on PBS system configuration, these *resource* parameters 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 Grid Engine/Oracle Grid Engine/Son of Grid Engine (SGE) and Univa Grid Engine (UGE) 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 To make sure all tasks are assign to a single node, specify --nodes=1, e.g. sbatch --nodes=1 --ntasks=16 hello.sh
- "custom" - If option parallelly.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. It is safe for this custom function to call availableCores(); if done, the custom function will *not* be recursively called.

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 some machines might have a limited number of cores, or the R process runs in a container or a cgroup that only provides a small number of cores. In such cases:

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

may return zero, which is often not intended and is likely to give an error downstream. Instead, use:

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

to put aside one of the cores from being used. Regardless how many cores you put aside, this function is guaranteed to return at least one core.

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(parallelly.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 <- availableCores() - 1      ## ncores = 0
ncores <- availableCores(omit = 1)  ## ncores = 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)
  ncores <- min(as.integer(0.75 * ncores), 4L)
  max(1L, ncores)
})
message(paste("Number of cores available:", availableCores()))

## Use 50% of the cores according to availableCores(), e.g.
## allocated by a job scheduler or cgroups.
## Note that it is safe to call availableCores() here.
options(parallelly.availableCores.custom = function() {
  0.50 * parallelly::availableCores()
})
message(paste("Number of cores available:", availableCores()))

## End(Not run)
```

`availableWorkers` *Get Set of Available Workers*

Description

Get Set of Available Workers

Usage

```
availableWorkers(
  constraints = NULL,
```

```
  methods = getOption2("parallelly.availableWorkers.methods", c("mc.cores",
   "BiocParallel", "_R_CHECK_LIMIT_CORES_", "Bioconductor", "LSF", "PJM", "PBS", "SGE",
   "Slurm", "custom", "cgroups.cpuset", "cgroups.cpuquota", "cgroups2.cpu.max", "nproc",
    "system", "fallback")),
  na.rm = TRUE,
  logical = getOption2("parallelly.availableCores.logical", TRUE),
  default = getOption2("parallelly.localhost.hostname", "localhost"),
  which = c("auto", "min", "max", "all")
)
```

Arguments

constraints	An optional character specifying under what constraints ("purposes") we are requesting the values. Using constraints = "connections", will append "connections" to the methods argument.
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" (default), 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:

- "LSF" - Query LSF/OpenLava environment variable LSB_HOSTS.
- "PJM" - Query Fujitsu Technical Computing Suite (that we choose to shorten as "PJM") the hostname file given by environment variable PJM_O_NODEINF. The PJM_O_NODEINF file lists the hostnames of the nodes allotted. This function returns those hostnames each repeated availableCores() times, where availableCores() reflects PJM_VNODE_CORE. For example, for pjsub -L vnode=2 -L vnode-core=8 hello.sh, the PJM_O_NODEINF file gives two hostnames, and PJM_VNODE_CORE gives eight cores per host, resulting in a character vector of 16 hostnames (for two unique hostnames).
- "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 Grid Engine/Oracle Grid Engine/Son of Grid Engine (SGE) and Univa Grid Engine (UGE) 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.

- "Slurm" - Query Slurm environment variable SLURM_JOB_NODELIST (fallback to legacy SLURM_NODELIST) and parse set of nodes. Then query Slurm environment variable SLURM_JOB_CPUS_PER_NODE (fallback SLURM_TASKS_PER_NODE) to infer how many CPU cores Slurm have allotted to each of the nodes. If SLURM_CPUS_PER_TASK is set, which is always a scalar, then that is respected too, i.e. if it is smaller, then that is used for all nodes. For example, if SLURM_NODELIST="n1,n[03-05]" (expands to c("n1", "n03", "n04", "n05")) and SLURM_JOB_CPUS_PER_NODE="2(x2),3,2" (expands to c(2, 2, 3, 2)), then c("n1", "n1", "n03", "n03", "n04", "n04", "n04", "n05", "n05") is returned. If in addition, SLURM_CPUS_PER_TASK=1, which can happen depending on hyperthreading configurations on the Slurm cluster, then c("n1", "n03", "n04", "n05") is returned.
- "custom" - If option parallelly.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. It is safe for this custom function to call availableWorkers(); if done, the custom function will *not* be recursively called.

Value

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

Known limitations

availableWorkers(methods = "Slurm") will expand SLURM_JOB_NODELIST using scontrol show hostnames "$SLURM_JOB_NODELIST", if available. If not available, then it attempts to parse the compressed nodelist based on a best-guess understanding on what the possible syntax may be. One known limitation is that "multi-dimensional" ranges are not supported, e.g. "a[1-2]b[3-4]" is expanded by scontrol to c("a1b3", "a1b4", "a2b3", "a2b4"). If scontrol is not available, then any components that failed to be parsed are dropped with an informative warning message. If no components could be parsed, then the result of methods = "Slurm" will be empty.

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)

## Not run:
## A 50% random subset of the available workers.
## Note that it is safe to call availableWorkers() here.
options(parallelly.availableWorkers.custom = function() {
  workers <- parallelly::availableWorkers()
  sample(workers, size = 0.50 * length(workers))
})
message(paste("Available workers:",
        paste(sQuote(availableWorkers()), collapse = ", ")))

## End(Not run)
```

`freePort` *Find a TCP port that can be opened*

Description

Find a TCP port that can be opened

Usage

```
freePort(ports = 1024:65535, default = "random", randomize = TRUE)
```

Arguments

`ports`	(integer vector, or character string) Zero or more TCP ports in [0, 65535] to scan. If `"random"`, then a random set of ports is considered. If `"auto"`, then the port given by environment variable `R_PARALLEL_PORT` is used, which may also specify `random`.
`default`	(integer) `NA_integer_` or a port to returned if an available port could not be found. If `"first"`, then `ports[1]`. If `"random"`, then a random port among `ports` is used. If `length(ports) == 0`, then `NA_integer_`.
`randomize`	(logical) If TRUE, `ports` is randomly shuffled before searched. This shuffle does *not* forward the RNG seed.

Value

Returns an integer representing the first port among `ports` that can be opened. If none can be opened, then `default` is returned. If port querying is not supported, as in R (< 4.0.0), then `default` is returned.

`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)
```

isForkedChild *Checks whether or not we are running in a forked child process*

Description

Checks whether or not we are running in a forked child process

Usage

```
isForkedChild()
```

Details

Examples of setups and functions that rely on *forked* parallelization are `parallel::makeCluster(n, type = "FORK")`, `parallel::mclapply()`, and `future::plan("multicore")`.

Value

(logical) Returns TRUE if the running in a forked child process, otherwise FALSE.

isForkedNode *Checks whether or not a Cluster Node Runs in a Forked Process*

Description

Checks whether or not a Cluster Node Runs in a Forked Process

Usage

```
isForkedNode(node, ...)
```

Arguments

`node`	A cluster node of class `SOCKnode` or `SOCK0node`.
`...`	Not used.

Value

(logical) Returns TRUE if the cluster node is running in a forked child process and FALSE if it does not. If it cannot be inferred, NA is returned.

isLocalhostNode *Checks whether or not a Cluster Node Runs on Localhost*

Description

Checks whether or not a Cluster Node Runs on Localhost

Usage

```
isLocalhostNode(node, ...)
```

Arguments

node	A cluster node of class SOCKnode or SOCK0node.
...	Not used.

Value

(logical) Returns TRUE if the cluster node is running on the current machine and FALSE if it runs on another machine. If it cannot be inferred, NA is returned.

isNodeAlive *Check whether or not the cluster nodes are alive*

Description

Check whether or not the cluster nodes are alive

Usage

```
isNodeAlive(x, ...)
```

Arguments

x	A cluster or a cluster node ("worker").
...	Not used.

Details

This function works by checking whether the cluster node process is running or not. This is done by querying the system for its process ID (PID), which is registered by `makeClusterPSOCK()` when the node starts. If the PID is not known, the NA is returned. On Unix and macOS, the PID is queried using `tools::pskill()` with fallback to `system("ps")`. On MS Windows, `system2("tasklist")` is used, which may take a long time if there are a lot of processes running. For details, see the *internal* `pid_exists()` function.

Value

A logical vector of length `length(x)` with values FALSE, TRUE, and NA. If it can be established that the process for a cluster node is running, then TRUE is returned. If it does not run, then FALSE is returned. If neither can be inferred, for instance because the worker runs on a remote machine, then NA is returned.

See Also

Use `parallel::stopCluster()` to shut down cluster nodes. If that's not sufficient, `killNode()` may be attempted.

Examples

```
cl <- makeClusterPSOCK(2)

## Check if cluster nodes #2 is alive
print(isNodeAlive(cl[[2]]))

## Check all nodes
print(isNodeAlive(cl))
```

`killNode` *Terminate one or more cluster nodes using process signaling*

Description

Terminate one or more cluster nodes using process signaling

Usage

```
killNode(x, signal = tools::SIGTERM, ...)
```

Arguments

`x`	cluster or cluster node to terminate.
`signal`	An integer that specifies the signal level to be sent to the parallel R process. It's only `tools::SIGINT` (2) and `tools::SIGTERM` (15) that are supported on all operating systems (i.e. Unix, macOS, and MS Windows). All other signals are platform specific, cf. `tools::pskill()`.
`...`	Not used.

Details

Note that the preferred way to terminate a cluster is via `parallel::stopCluster()`, because it terminates the cluster nodes by kindly asking each of them to nicely shut themselves down. Using `killNode()` is a much more sever approach. It abruptly terminates the underlying R process, possibly without giving the parallel worker a chance to terminate gracefully. For example, it might get terminated in the middle of writing to file.

`tools::pskill()` is used to send the signal to the R process hosting the parallel worker.

Value

TRUE if the signal was successfully applied, FALSE if not, and NA if signaling is not supported on the specific cluster or node. *Warning*: With R (< 3.5.0), NA is always returned. This is due to a bug in R (< 3.5.0), where the signaling result cannot be trusted.

Known limitations

This function works only with cluster nodes of class `RichSOCKnode`, which were created by `makeClusterPSOCK()`. It does not work when using `parallel::makeCluster()` and friends.

Currently, it's only possible to send signals to parallel workers, that is, cluster nodes, that run on the local machine. If attempted to use `killNode()` on a remote parallel workers, `NA` is returned and an informative warning is produced.

See Also

Use `isNodeAlive()` to check whether one or mode cluster nodes are alive.

Examples

```
cl <- makeClusterPSOCK(2)
print(isNodeAlive(cl))  ## [1] TRUE TRUE

res <- killNode(cl)
print(res)

## It might take a moment before the background
## workers are shutdown after having been signaled
Sys.sleep(1.0)

print(isNodeAlive(cl))  ## [1] FALSE FALSE
```

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 = getOption2("parallelly.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 = getOption2("parallelly.makeNodePSOCK.tries", 3L),
  delay = getOption2("parallelly.makeNodePSOCK.tries.delay", 15),
  validate = getOption2("parallelly.makeNodePSOCK.validate", TRUE),
  verbose = getOption2("parallelly.debug", FALSE)
)

makeNodePSOCK(
  worker = getOption2("parallelly.localhost.hostname", "localhost"),
  master = NULL,
  port,
  connectTimeout = getOption2("parallelly.makeNodePSOCK.connectTimeout", 2 * 60),
  timeout = getOption2("parallelly.makeNodePSOCK.timeout", 30 * 24 * 60 * 60),
  rscript = NULL,
  homogeneous = NULL,
```

```
  rscript_args = NULL,
  rscript_envs = NULL,
  rscript_libs = NULL,
  rscript_startup = NULL,
  rscript_sh = c("auto", "cmd", "sh"),
  default_packages = c("datasets", "utils", "grDevices", "graphics", "stats", if
    (methods) "methods"),
  methods = TRUE,
 socketOptions = getOption2("parallelly.makeNodePSOCK.socketOptions", "no-delay"),
  useXDR = getOption2("parallelly.makeNodePSOCK.useXDR", FALSE),
  outfile = "/dev/null",
  renice = NA_integer_,
  rshcmd = getOption2("parallelly.makeNodePSOCK.rshcmd", NULL),
  user = NULL,
  revtunnel = NA,
  rshlogfile = NULL,
  rshopts = getOption2("parallelly.makeNodePSOCK.rshopts", NULL),
  rank = 1L,
  manual = FALSE,
  dryrun = FALSE,
  quiet = FALSE,
 setup_strategy = getOption2("parallelly.makeNodePSOCK.setup_strategy", "parallel"),
  action = c("launch", "options"),
  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_PARALLELLY_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. Arguments `tries` and `delay` are used only when `setup_strategy == "sequential`.

`validate` If TRUE (default), 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 or unset on worker at startup, e.g. `rscript_envs = c(FOO = "3.14", "HOME", "UNKNOWN", UNSETME = NA_character_)`. 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()`. An named element with value `NA_character_` will cause that variable to be unset, which is done via `Sys.unsetenv()`.

`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.

`rscript_sh` The type of shell used where `rscript` is launched, which should be `"sh"` is launched via a POSIX shell and `"cmd"` if launched via an MS Windows shell. This controls how shell command-line options are quoted, via `shQuote(...,`

type = rscript_sh). If "auto" (default), and the cluster node is launched locally, then it is set to "sh" or "cmd" according to the current platform. If launched remotely, then it is set to "sh" based on the assumption remote machines typically launch commands via SSH in a POSIX shell.

default_packages
: A character vector or NULL that controls which R packages are attached on each cluster node during startup. An asterisk ("*") resolves to getOption("defaultPackages") *on the current machine*. If NULL, then the default set of packages R are attached.

methods
: If TRUE (default), then the **methods** package is also loaded. This is argument exists for legacy reasons due to how Rscript worked in R (< 3.5.0).

socketOptions
: A character string that sets R option socketOptions on the worker.

useXDR
: If FALSE (default), the communication between master and workers, which is binary, will use small-endian (faster), otherwise big-endian ("XDR"; slower).

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. If NA, then TRUE or FALSE is inferred from inspection of rshcmd[1]. 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 command-line option '-E out.log'.* For example, PuTTY's plink does *not* support this option, and any attempts to specify rshlogfile will cause the SSH connection to fail.

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.

quiet
: If TRUE, then no output will be produced other than that from using verbose = TRUE.

setup_strategy
: If "parallel" (default), the workers are set up concurrently, one after the other. If "sequential", they are set up sequentially.

action
: This is an internal argument.

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 `parallelly.makeNodePSOCK.rshcmd`. If that is not set, then the default is to use `ssh` on Unix-like systems, including macOS as well as Windows 10. On older MS Windows versions, which does not have a built-in `ssh` client, the default is to use (i) `plink` from the PuTTY project, and then (ii) the `ssh` client that is distributed with RStudio.

PuTTY puts itself on Windows' 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. To use this SSH client outside of RStudio, set `RSTUDIO_MSYS_SSH` accordingly.

You can override the default set of SSH clients that are searched for by specifying them in argument `rshcmd` or via option `parallelly.makeNodePSOCK.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.

Additional SSH command-line options may be specified via argument `rshopts`, which defaults to option `parallelly.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

If SSH is used, which is inferred from `rshcmd[1]`, then the default is to use reverse SSH tunneling (`revtunnel = TRUE`), otherwise not (`revtunnel = FALSE`). Using reverse SSH tunneling, avoids complications from otherwise having to configure port forwarding in firewalls, which often requires static IP address as well as privileges to edit the firewall on your outgoing router, 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.

When specified, argument `rscript` should be a character vector with one or more elements. Any asterisk (`"*"`) will be resolved to the above default `homogeneous`-dependent `Rscript` path. 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.

If R stalls when setting up a cluster of local workers, then it might be that you have a virtual private network (VPN) enabled that is configured to prevent you from connecting to localhost. To verify that this is the case, call the following from the terminal:

```
{local}$ ssh localhost "date"
```

This also freezed if the VPN intercepts connections to localhost. If this happens, try also:

```
{local}$ ssh 127.0.0.1 "date"
```

In rare cases, 127.0.0.1 might work when localhost does not. If the latter works, setting R option:

```
options(parallelly.localhost.hostname = "127.0.0.1")
```

should solve it (the default is "localhost"). You can set this automatically when R starts by adding it to your ~/.Rprofile startup file. Alternatively, set environment variable R_PARALLELLY_LOCALHOST_HOSTNAME=127.0.0. in your ~/.Renviron file.

If using 127.0.0.1 did not work around the problem, check your VPN settings and make sure it allows connections to localhost or 127.0.0.1.

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 conn This is because R (>= 3.6.0) uses serialization format version 3 by default 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().

For package developers

When creating a cluster object, for instance via parallel::makeCluster() or parallelly::makeClusterPSOCK(), in a package help example, in a package vignette, or in a package test, we must *remember to stop the cluster at the end of all examples(*), vignettes, and unit tests*. This is required in order to not leave behind stray parallel cluster workers after our main R session terminates. On Linux and macOS, the operating system often takes care of terminating the worker processes if we forget, but on MS Windows such processes will keep running in the background until they time out themselves, which takes 30 days (sic!).

`R CMD check --as-cran` will indirectly detect these stray worker processes on MS Windows when running R (>= 4.3.0). They are detected, because they result in placeholder `Rscript<hexcode>` files being left behind in the temporary directory. The check NOTE to look out for (only in R (>= 4.3.0)) is:

```
* checking for detritus in the temp directory ... NOTE
Found the following files/directories:
  'Rscript1058267d0c10' 'Rscriptbd4267d0c10'
```

Those `Rscript<hexcode>` files are from background R worker processes, which almost always are parallel `cluster`:s that we forgot to stop at the end. To stop all `cluster` workers, use `parallel::stopCluster()` at the end of your examples(*), vignettes, and package tests for every `cluster` object that is created.

(*) Currently, examples are excluded from the detritus checks. This was validated with R-devel revision 82991 (2022-10-02).

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, quiet = 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, quiet = 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, quiet = TRUE
)

## EXAMPLE: Three remote workers 'n1', 'n2', and 'n3' that can only be
## accessed via jumphost 'login.remote.org'
workers <- c("n1", "n2", "n1")
cl <- makeClusterPSOCK(
  workers,
  rshopts = c("-J", "login.remote.org"),
  homogeneous = FALSE,
  dryrun = TRUE, quiet = TRUE
)

## EXAMPLE: Remote workers with specific setup
## Setup of remote worker with more detailed control on
## authentication and reverse SSH tunneling
```

```
cl <- makeClusterPSOCK(
  "remote.server.org", user = "johnny",
  ## Manual configuration of reverse SSH tunneling
  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"),
  dryrun = TRUE, quiet = 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, quiet = 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, quiet = 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:::.workRSOCK()"))
  ),
  dryrun = TRUE, quiet = TRUE
)


## EXAMPLE: One worker running in Wine for Linux on the local machine
## To install R for MS Windows in Wine, do something like:
##   winecfg  # In GUI, set 'Windows version' to 'Windows 10'
##   wget https://cran.r-project.org/bin/windows/base/R-4.1.2-win.exe
##   wine R-4.1.2-win.exe /SILENT
## Prevent packages from being installed to R's system library:
##   chmod ugo-w "$HOME/.wine/drive_c/Program Files/R/R-4.1.2/library/"
## Verify it works:
##   wine "C:/Program Files/R/R-4.1.2/bin/x64/Rscript.exe" --version
cl <- makeClusterPSOCK(1L,
  rscript = c(
    ## Silence Wine warnings
    "WINEDEBUG=fixme-all",
    ## Don't pass LC_*** environments from Linux to Wine
    sprintf("%s=", grep("LC_", names(Sys.getenv()), value = TRUE)),
    "wine",
    "C:/Program Files/R/R-4.1.2/bin/x64/Rscript.exe"
  ),
  dryrun = TRUE, quiet = TRUE
)


## EXAMPLE: Launch 124 workers on MS Windows 10, where half are
## running on CPU Group #0 and half on CPU Group #1.
## (https://lovickconsulting.com/2021/11/18/
##  running-r-clusters-on-an-amd-threadripper-3990x-in-windows-10-2/)
ncores <- 124
cpu_groups <- c(0, 1)
cl <- lapply(cpu_groups, FUN = function(cpu_group) {
    parallelly::makeClusterPSOCK(ncores %/% length(cpu_groups),
      rscript = I(c(
        Sys.getenv("COMSPEC"), "/c", "start", "/B",
        "/NODE", cpu_group, "/AFFINITY", "0xFFFFFFFFFFFFFFFE",
        "*"
      )),
      dryrun = TRUE, quiet = TRUE
    )
})
## merge the two 62-node clusters into one with 124 nodes
cl <- do.call(c, cl)


## EXAMPLE: Remote worker running on AWS
## Launching worker on Amazon AWS EC2 running one of the
## Amazon Machine Images (AMI) provided by RStudio
```

```
## (https://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, quiet = 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, quiet = 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, quiet = 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, quiet = TRUE
)


## EXAMPLE: The 'Fujitsu Technical Computing Suite' is a high-performance
## compute (HPC) job scheduler where one can request compute resources on
## multiple nodes, each running multiple cores.  For example,
##
##   pjsub -L vnode=3 -L vnode-core=18 script.sh
##
## reserves 18 cores on three nodes. The job script runs on the first
## with enviroment variables set to infer the other nodes, resulting in
## availableWorkers() to return 3 * 18 workers. When the HPC environment
## does not support SSH between compute nodes, one can use the 'pjrsh'
## command to launch the parallel workers.
cl <- makeClusterPSOCK(
  availableWorkers(),
  rshcmd = "pjrsh",
  dryrun = TRUE, quiet = TRUE
)
```

`parallelly.options` *Options Used by the 'parallelly' Package*

Description

Below are the R options and environment variables that are used by the **parallelly** package and packages enhancing it.

WARNING: Note that the names and the default values of these options may change in future versions of the package. Please use with care until further notice.

Backward compatibility with the future package

The functions in the **parallelly** package originates from the **future** package. Because they are widely used within the future ecosystem, we need to keep them backward compatible for quite a long time, in order for all existing packages and R scripts to have time to adjust. This also goes for the R options and the environment variables used to configure these functions. All options and environment variables used here have prefixes `parallelly.` and `R_PARALLELLY_`, respectively. Because of the backward compatibility with the **future** package, the same settings can also be controlled by options and environment variables with prefixes `future.` and `R_FUTURE_` until further notice, e.g. setting option `future.availableCores.fallback=1` is the same as setting option `parallelly.availableCores.fallback=1`, and setting environment variable `R_FUTURE_AVAILABLECORES_FALLBACK=1` is the same as setting `R_PARALLELLY_AVAILABLECORES_FALLBACK=1`.

Configuring number of parallel workers

The below R options and environment variables control the default results of `availableCores()` and `availableWorkers()`.

`parallelly.availableCores.logical`: (logical) The default value of argument `logical` as used by `availableCores()`, `availableWorkers()`, and `availableCores()` for querying `parallel::detectCores(logic = logical)`. The default is `TRUE` just like it is for `parallel::detectCores()`.

`parallelly.availableCores.methods`: (character vector) Default lookup methods for `availableCores()`. (Default: `c("system", "cgroups.cpuset", "cgroups.cpuquota", "cgroups2.cpu.max", "nproc", "mc.cores", "BiocParallel", "_R_CHECK_LIMIT_CORES_", "Bioconductor", "LSF", "PJM", "PBS", "SGE", "Slurm", "fallback", "custom")`)

`parallelly.availableCores.custom`: (function) If set and a function, then this function will be called (without arguments) by `availableCores()` where its value, coerced to an integer, is interpreted as a number of cores.

`parallelly.availableCores.fallback`: (integer) Number of cores to use when no core-specifying settings are detected other than `"system"` and `"nproc"`. This options makes it possible to set the default number of cores returned by `availableCores()` / `availableWorkers()` yet allow users and schedulers to override it. In multi-tenant environment, such as HPC clusters, it is useful to set environment variable `R_PARALLELLY_AVAILABLECORES_FALLBACK` to `1`, which will set this option when the package is loaded.

`parallelly.availableCores.system`: (integer) Number of "system" cores used instead of what is reported by `availableCores(which = "system")`. This option allows you to effectively override what `parallel::detectCores()` reports the system has.

`parallelly.availableCores.min`: (integer) The minimum number of cores `availableCores()` is allowed to return. This can be used to force multiple cores on a single-core environment. If this is limit is applied, the names of the returned value are appended with an asterisk (`*`). (Default: `1L`)

`parallelly.availableCores.omit`: (integer) Number of cores to set aside, i.e. not to include.

`parallelly.availableWorkers.methods`: (character vector) Default lookup methods for `availableWorkers()`. (Default: `c("mc.cores", "BiocParallel", "_R_CHECK_LIMIT_CORES_", "Bioconductor",`

"LSF", "PJM", "PBS", "SGE", "Slurm", "custom", "cgroups.cpuset", "cgroups.cpuquota", "cgroups2.cpu.max", "nproc", "system", "fallback"))

`parallelly.availableWorkers.custom`: (function) If set and a function, then this function will be called (without arguments) by `availableWorkers()` where its value, coerced to a character vector, is interpreted as hostnames of available workers.

Configuring forked parallel processing

The below R options and environment variables control the default result of `supportsMulticore()`.

`parallelly.fork.enable`: (logical) Enable or disable *forked* processing. If `FALSE`, multicore futures becomes sequential futures. If `NA`, or not set (the default), the a set of best-practices rules decide whether should be supported or not.

`parallelly.supportsMulticore.unstable`: (character) Controls whether a warning should be produced or not whenever multicore processing is automatically disabled because the environment in which R runs is considered unstable for forked processing, e.g. in the RStudio environment. If `"warn"` (default), then an informative warning is produces the first time 'multicore' or 'multiprocess' futures are used. If `"quiet"`, no warning is produced.

Configuring setup of parallel PSOCK clusters

The below R options and environment variables control the default results of `makeClusterPSOCK()` and its helper function `makeNodePSOCK()` that creates the individual cluster nodes.

`parallelly.makeNodePSOCK.setup_strategy`: (character) If `"parallel"` (default), the PSOCK cluster nodes are set up concurrently. If `"sequential"`, they are set up sequentially.

`parallelly.makeNodePSOCK.validate`: (logical) If TRUE (default), 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.

`parallelly.makeNodePSOCK.connectTimeout`: (numeric) The maximum time (in seconds) allowed for each socket connection between the master and a worker to be established (defaults to 2*60 seconds = 2 minutes).

`parallelly.makeNodePSOCK.timeout`: (numeric) The maximum time (in seconds) allowed to pass without the master and a worker communicate with each other (defaults to 30*24*60*60 seconds = 30 days).

`parallelly.makeNodePSOCK.useXDR`: (logical) If FALSE (default), the communication between master and workers, which is binary, will use small-endian (faster), otherwise big-endian ("XDR"; slower).

`parallelly.makeNodePSOCK.socketOptions`: (character string) If set to another value than `"NULL"`, then option `socketOptions` is set to this value on the workers during startup. See `base::socketConnection()` for details. (defaults to `"no-delay"`)

`parallelly.makeNodePSOCK.rshcmd`: (character vector) The command to be run on the master to launch a process on another host.

`parallelly.makeNodePSOCK.rshopts`: (character vector) Addition command-line options appended to `rshcmd`. These arguments are only applied when connecting to non-localhost machines.

`parallelly.makeNodePSOCK.tries`: (integer) The maximum number of attempts done to launch each node. Only used when setting up cluster nodes using the sequential strategy.

`parallelly.makeNodePSOCK.tries.delay`**:** (numeric) The number of seconds to wait before trying to launch a cluster node that failed to launch previously. Only used when setting up cluster nodes using the sequential strategy.

Options for debugging

`parallelly.debug`**:** (logical) If `TRUE`, extensive debug messages are generated. (Default: `FALSE`)

Environment variables that set R options

All of the above R `parallelly.*` options can be set by corresponding environment variables `R_PARALLELLY_*` *when the* **parallelly** *package is loaded.* For example, if `R_PARALLELLY_MAKENODEPSOCK_SETUP_STRATEGY = "sequential"`, then option `parallelly.makeNodePSOCK.setup_strategy` is set to `"sequential"` (character). Similarly, if `R_PARALLELLY_AVAILABLECORES_FALLBACK = "1"`, then option `parallelly.availableCores.fa` is set to 1 (integer).

See Also

To set R options when R starts (even before the **parallelly** package is loaded), see the Startup help page. The **startup** package provides a friendly mechanism for configuring R's startup process.

Examples

```
# Set an R option:
options(parallelly.availableCores.fallback = 1L)
```

---

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

While R supports forked processing on Unix-like operating system such as Linux and macOS, it does not on the Microsoft Windows operating system.

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 `parallelly.supportsMulticore.unstable`, or environment variable `R_PARALLELLY_SUPPORTSMULTICORE_UNSTABLE` to `"quiet"`.

Enable or disable forked processing

It is possible to disable forked processing for futures by setting R option `parallelly.fork.enable` to `FALSE`. Alternatively, one can set environment variable `R_PARALLELLY_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

as.cluster, 2
autoStopCluster, 3
autoStopCluster(), *18*, *20*
availableConnections, 4
availableCores, 5, *32*
availableCores(), *9*, *10*, *32*
availableWorkers, 8
availableWorkers(), *7*, *32*, *33*

base::closeAllConnections(), *12*
base::getConnection(), *12*, *13*
base::showConnections(), *4*, *12*, *13*
base::shQuote(), *24*
base::socketConnection(), *33*
base::sys.save.image(), *13*

c.cluster *(*as.cluster*)*, 2
connection, *12*
connectionId *(*isConnectionValid*)*, 12
connections, *4*, *12*

detectCores, *5*, *6*

freeConnections *(*availableConnections*)*, 4
freeConnections(), *6*
freePort, 11
future.availableCores.custom *(*parallelly.options*)*, 31
future.availableCores.fallback *(*parallelly.options*)*, 31
future.availableCores.methods *(*parallelly.options*)*, 31
future.availableCores.system *(*parallelly.options*)*, 31
future.availableWorkers.custom *(*parallelly.options*)*, 31
future.availableWorkers.methods *(*parallelly.options*)*, 31
future.fork.enable *(*parallelly.options*)*, 31
future.supportsMulticore.unstable *(*parallelly.options*)*, 31

isConnectionValid, 12
isForkedChild, 14
isForkedNode, 14
isLocalhostNode, 15
isNodeAlive, 15
isNodeAlive(), *17*

killNode, 16
killNode(), *16*

makeCluster, *18*
makeClusterMPI, 18
makeClusterPSOCK, 19
makeClusterPSOCK(), *3*, *4*, *15*, *17*, *18*, *33*
makeNodePSOCK *(*makeClusterPSOCK*)*, 19
makeNodePSOCK(), *33*
makePSOCKcluster, *19*
mc.cores, *6*
mclapply, *6*

parallel::detectCores(), *32*
parallel::makeCluster(), *3*, *4*, *17*, *18*
parallel::stopCluster(), *16*, *17*, *27*
parallelly.availableConnections.tries, *4*
parallelly.availableCores.custom, *7*
parallelly.availableCores.custom *(*parallelly.options*)*, 31
parallelly.availableCores.fallback *(*parallelly.options*)*, 31
parallelly.availableCores.methods *(*parallelly.options*)*, 31
parallelly.availableCores.min *(*parallelly.options*)*, 31
parallelly.availableCores.omit *(*parallelly.options*)*, 31
parallelly.availableCores.system *(*parallelly.options*)*, 31

parallelly.availableWorkers.custom, *10*
parallelly.availableWorkers.custom (parallelly.options), 31
parallelly.availableWorkers.methods (parallelly.options), 31
parallelly.debug (parallelly.options), 31
parallelly.fork.enable (parallelly.options), 31
parallelly.makeNodePSOCK.connectTimeout (parallelly.options), 31
parallelly.makeNodePSOCK.rshcmd, *23*
parallelly.makeNodePSOCK.rshcmd (parallelly.options), 31
parallelly.makeNodePSOCK.rshopts (parallelly.options), 31
parallelly.makeNodePSOCK.setup_strategy (parallelly.options), 31
parallelly.makeNodePSOCK.socketOptions (parallelly.options), 31
parallelly.makeNodePSOCK.timeout (parallelly.options), 31
parallelly.makeNodePSOCK.tries (parallelly.options), 31
parallelly.makeNodePSOCK.useXDR (parallelly.options), 31
parallelly.makeNodePSOCK.validate (parallelly.options), 31
parallelly.options, 31
parallelly.supportsMulticore.unstable (parallelly.options), 31
pid_exists(), *15*

R_FUTURE_AVAILABLECORES_FALLBACK (parallelly.options), 31
R_FUTURE_AVAILABLECORES_SYSTEM (parallelly.options), 31
R_FUTURE_FORK_ENABLE (parallelly.options), 31
R_FUTURE_SUPPORTSMULTICORE_UNSTABLE (parallelly.options), 31
R_PARALLELLY_AVAILABLECORES_FALLBACK (parallelly.options), 31
R_PARALLELLY_AVAILABLECORES_MIN (parallelly.options), 31
R_PARALLELLY_AVAILABLECORES_OMIT (parallelly.options), 31
R_PARALLELLY_AVAILABLECORES_SYSTEM (parallelly.options), 31
R_PARALLELLY_FORK_ENABLE (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_CONNECTTIMEOUT (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_RSHCMD (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_RSHOPTS (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_SETUP_STRATEGY (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_SOCKETOPTIONS (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_TIMEOUT (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_TRIES (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_TRIES_DELAY (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_USEXDR (parallelly.options), 31
R_PARALLELLY_MAKENODEPSOCK_VALIDATE (parallelly.options), 31
R_PARALLELLY_SUPPORTSMULTICORE_UNSTABLE (parallelly.options), 31

Startup, *34*
stderr, *22*
stderr(), *4*
stdin(), *4*
stdout, *22*
stdout(), *4*
stopCluster, *3*, *18*, *20*
stopCluster(), *18*
supportsMulticore, 34
supportsMulticore(), *33*

tools::pskill(), *15–17*

utils::capture.output(), *4*