Package 'batchtools'

August 16, 2018

Title Tools for Computation on Batch Systems

Version 0.9.11

Description As a successor of the packages 'BatchJobs' and 'BatchExperiments', this package provides a parallel implementation of the Map function for high performance computing systems managed by schedulers 'IBM Spectrum LSF' (<https://www.ibm.com/us-en/marketplace/hpc-workload-management>), 'OpenLava' (<http://www.openlava.org/>), 'Univa Grid Engine'/'Oracle Grid Engine' (<http://www.univa.com/>), 'Slurm' (<http://slurm.schedmd.com/>), 'TORQUE/PBS' (<http://www.adaptivecomputing.com/products/open-source/torque/>), or 'Docker Swarm' (<https://docs.docker.com/swarm/>).
A multicore and socket mode allow the parallelization on a local machines, and multiple machines can be hooked up via SSH to create a makeshift cluster. Moreover, the package provides an abstraction mechanism to define large-scale computer experiments in a well-organized and reproducible way.

License LGPL-3

URL https://github.com/mllg/batchtools

BugReports https://github.com/mllg/batchtools/issues

NeedsCompilation yes

ByteCompile yes

Encoding UTF-8

Depends R (>= 3.0.0), data.table (>= 1.11.2)

Imports backports (>= 1.1.2), base64url (>= 1.1), brew, checkmate (>= 1.8.5), digest (>= 0.6.9), fs (>= 1.2.0), parallel, progress (>= 1.1.1), R6, rappdirs, stats, stringi, utils, withr (>= 2.0.0)

Suggests debugme, doParallel, doMPI, e1071, foreach, future, future.batchtools, knitr, parallelMap, ranger, rmarkdown, rpart, snow, testthat, tibble

VignetteBuilder knitr

RoxygenNote 6.1.0

Author Michel Lang [cre, aut] (<https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut],
Dirk Surmann [ctb] (<https://orcid.org/0000-0003-0873-137X>)

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2018-08-16 11:40:03 UTC

R topics documented:

`batchtools-package` *batchtools: Tools for Computation on Batch Systems*

Description

For bug reports and feature requests please use the tracker: https://github.com/mllg/batchtools.

Package options

`batchtools.verbose` Verbosity. Set to `FALSE` to suppress info messages and progress bars.

`batchtools.progress` Progress bars. Set to `FALSE` to disable them.

`batchtools.timestamps` Add time stamps to log output. Set to `FALSE` to disable them.

Furthermore, you may enable a debug mode using the **debugme** package by setting the environment variable "DEBUGME" to "batchtools" before loading **batchtools**.

Author(s)

Maintainer: Michel Lang <michellang@gmail.com> (0000-0001-9754-0393)

Authors:

- Bernd Bischl <bernd_bischl@gmx.de>

Other contributors:

- Dirk Surmann <surmann@statistik.tu-dortmund.de> (0000-0003-0873-137X) [contributor]

See Also

Useful links:

- https://github.com/mllg/batchtools
- Report bugs at https://github.com/mllg/batchtools/issues

---

addAlgorithm *Define Algorithms for Experiments*

Description

Algorithms are functions which get the codedata part as well as the problem instance (the return value of the function defined in Problem) and return an arbitrary R object.

This function serializes all components to the file system and registers the algorithm in the ExperimentRegistry.

removeAlgorithm removes all jobs from the registry which depend on the specific algorithm. reg$algorithms holds the IDs of already defined algorithms.

Usage

```
addAlgorithm(name, fun = NULL, reg = getDefaultRegistry())

removeAlgorithms(name, reg = getDefaultRegistry())
```

Arguments

name	[character(1)] Unique identifier for the algorithm.
fun	[function] The algorithm function. The static problem part is passed as "data", the generated problem instance is passed as "instance" and the Job/Experiment as "job". Therefore, your function must have the formal arguments "job", "data" and "instance" (or dots ...). If you do not provide a function, it defaults to a function which just returns the instance.
reg	[ExperimentRegistry] Registry. If not explicitly passed, uses the last created registry.

Value

`Algorithm` . Object of class "Algorithm".

See Also

`Problem`, `addExperiments`

`addExperiments` *Add Experiments to the Registry*

Description

Adds experiments (parametrized combinations of problems with algorithms) to the registry and thereby defines batch jobs.

If multiple problem designs or algorithm designs are provided, they are combined via the Cartesian product. E.g., if you have two problems `p1` and `p2` and three algorithms `a1`, `a2` and `a3`, `addExperiments` creates experiments for all parameters for the combinations `(p1, a1)`, `(p1, a2)`, `(p1, a3)`, `(p2, a1)`, `(p2, a2)` and `(p2, a3)`.

Usage

```
addExperiments(prob.designs = NULL, algo.designs = NULL, repls = 1L,
  combine = "crossprod", reg = getDefaultRegistry())
```

Arguments

`prob.designs`	[named list of `data.frame`] Named list of data frames (or `data.table`). The name must match the problem name while the column names correspond to parameters of the problem. If `NULL`, experiments for all defined problems without any parameters are added.
`algo.designs`	[named list of `data.table` or `data.frame`] Named list of data frames (or `data.table`). The name must match the algorithm name while the column names correspond to parameters of the algorithm. If `NULL`, experiments for all defined algorithms without any parameters are added.
`repls`	[`integer(1)`] Number of replications for each experiment.
`combine`	[`character(1)`] How to combine the rows of a single problem design with the rows of a single algorithm design? Default is "crossprod" which combines each row of the problem design which each row of the algorithm design in a cross-product fashion. Set to "bind" to just `cbind` the tables of problem and algorithm designs where the shorter table is repeated if necessary.
`reg`	[`ExperimentRegistry`] Registry. If not explicitly passed, uses the last created registry.

Value

data.table with ids of added jobs stored in column "job.id".

Note

R's data.frame converts character vectors to factors by default which frequently resulted in problems using addExperiments. Therefore, this function will warn about factor variables if the following conditions hold:

1. The design is passed as a data.frame, not a data.table or tibble.
2. The option "stringsAsFactors" is not set or set to TRUE.

See Also

Other Experiment: removeExperiments, summarizeExperiments

Examples

```
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE)

# add first problem
fun = function(job, data, n, mean, sd, ...) rnorm(n, mean = mean, sd = sd)
addProblem("rnorm", fun = fun, reg = tmp)

# add second problem
fun = function(job, data, n, lambda, ...) rexp(n, rate = lambda)
addProblem("rexp", fun = fun, reg = tmp)

# add first algorithm
fun = function(instance, method, ...) if (method == "mean") mean(instance) else median(instance)
addAlgorithm("average", fun = fun, reg = tmp)

# add second algorithm
fun = function(instance, ...) sd(instance)
addAlgorithm("deviation", fun = fun, reg = tmp)

# define problem and algorithm designs
prob.designs = algo.designs = list()
prob.designs$rnorm = CJ(n = 100, mean = -1:1, sd = 1:5)
prob.designs$rexp = data.table(n = 100, lambda = 1:5)
algo.designs$average = data.table(method = c("mean", "median"))
algo.designs$deviation = data.table()

# add experiments and submit
addExperiments(prob.designs, algo.designs, reg = tmp)

# check what has been created
summarizeExperiments(reg = tmp)
unwrap(getJobPars(reg = tmp))
```

addProblem *Define Problems for Experiments*

Description

Problems may consist of up to two parts: A static, immutable part (data in addProblem) and a dynamic, stochastic part (fun in addProblem). For example, for statistical learning problems a data frame would be the static problem part while a resampling function would be the stochastic part which creates problem instance. This instance is then typically passed to a learning algorithm like a wrapper around a statistical model (fun in addAlgorithm).

This function serialize all components to the file system and registers the problem in the ExperimentRegistry.

removeProblem removes all jobs from the registry which depend on the specific problem. reg$problems holds the IDs of already defined problems.

Usage

```
addProblem(name, data = NULL, fun = NULL, seed = NULL,
  cache = FALSE, reg = getDefaultRegistry())

removeProblems(name, reg = getDefaultRegistry())
```

Arguments

name	[character(1)] Unique identifier for the problem.
data	[ANY] Static problem part. Default is NULL.
fun	[function] The function defining the stochastic problem part. The static part is passed to this function with name "data" and the Job/Experiment is passed as "job". Therefore, your function must have the formal arguments "job" and "data" (or dots ...). If you do not provide a function, it defaults to a function which just returns the data part.
seed	[integer(1)] Start seed for this problem. This allows the "synchronization" of a stochastic problem across algorithms, so that different algorithms are evaluated on the same stochastic instance. If the problem seed is defined, the seeding mechanism works as follows: (1) Before the dynamic part of a problem is instantiated, the seed of the problem + [replication number] - 1 is set, i.e. the first replication uses the problem seed. (2) The stochastic part of the problem is instantiated. (3) From now on the usual experiment seed of the registry is used, see ExperimentRegistry. If seed is set to NULL (default), the job seed is used to instantiate the problem and different algorithms see different stochastic instances of the same problem.

cache [logical(1)]
If TRUE and seed is set, problem instances will be cached on the file system. This assumes that each problem instance is deterministic for each combination of hyperparameter setting and each replication number. This feature is experimental.

reg [ExperimentRegistry]
Registry. If not explicitly passed, uses the last created registry.

Value

Problem . Object of class "Problem" (invisibly).

See Also

Algorithm, addExperiments

Examples

```
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE)
addProblem("p1", fun = function(job, data) data, reg = tmp)
addProblem("p2", fun = function(job, data) job, reg = tmp)
addAlgorithm("a1", fun = function(job, data, instance) instance, reg = tmp)
addExperiments(repls = 2, reg = tmp)

# List problems, algorithms and job parameters:
tmp$problems
tmp$algorithms
getJobPars(reg = tmp)

# Remove one problem
removeProblems("p1", reg = tmp)

# List problems and algorithms:
tmp$problems
tmp$algorithms
getJobPars(reg = tmp)
```

---

assertRegistry *assertRegistry*

Description

Assert that a given object is a batchtools registry. Additionally can sync the registry, check if it is writeable, or check if jobs are running. If any check fails, throws an error indicting the reason for the failure.

Usage

```
assertRegistry(reg, class = NULL, writeable = FALSE, sync = FALSE,
  running.ok = TRUE)
```

Arguments

reg	[Registry] The object asserted to be a Registry.
class	[character(1)] If NULL (default), reg must only inherit from class "Registry". Otherwise check that reg is of class class. E.g., if set to "Registry", a ExperimentRegistry would not pass.
writeable	[logical(1)] Check if the registry is writeable.
sync	[logical(1)] Try to synchronize the registry by including pending results from the file system. See syncRegistry.
running.ok	[logical(1)] If FALSE throw an error if jobs associated with the registry are currently running.

Value

TRUE invisibly.

---

batchExport *Export Objects to the Slaves*

Description

Objects are saved in subdirectory "exports" of the "file.dir" of reg. They are automatically loaded and placed in the global environment each time the registry is loaded or a job collection is executed.

Usage

```
batchExport(export = list(), unexport = character(0L),
  reg = getDefaultRegistry())
```

Arguments

export	[list] Named list of objects to export.
unexport	[character] Vector of object names to unexport.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

`data.table` with name and uri to the exported objects.

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)

# list exports
exports = batchExport(reg = tmp)
print(exports)

# add a job and required exports
batchMap(function(x) x^2 + y + z, x = 1:3, reg = tmp)
exports = batchExport(export = list(y = 99, z = 1), reg = tmp)
print(exports)

submitJobs(reg = tmp)
waitForJobs(reg = tmp)
stopifnot(loadResult(1, reg = tmp) == 101)

# Un-export z
exports = batchExport(unexport = "z", reg = tmp)
print(exports)
```

---

batchMap *Map Operation for Batch Systems*

Description

A parallel and asynchronous `Map`/`mapply` for batch systems. Note that this function only defines the computational jobs. The actual computation is started with `submitJobs`. Results and partial results can be collected with `reduceResultsList`, `reduceResults` or `loadResult`.

For a synchronous `Map`-like execution, see `btmapply`.

Usage

```
batchMap(fun, ..., args = list(), more.args = list(),
  reg = getDefaultRegistry())
```

Arguments

`fun`	[`function`] Function to map over arguments provided via `...`. Parameters given via `args` or `...` are passed as-is, in the respective order and possibly named. If the function has the named formal argument ".job", the `Job` is passed to the function on the slave.

`...` [ANY]
Arguments to vectorize over (list or vector). Shorter vectors will be recycled (possibly with a warning any length is not a multiple of the longest length). Mutually exclusive with `args`. Note that although it is possible to iterate over large objects (e.g., lists of data frames or matrices), this usually hurts the overall performance and thus is discouraged.

`args` [`list | data.frame`]
Arguments to vectorize over as (named) list or data frame. Shorter vectors will be recycled (possibly with a warning any length is not a multiple of the longest length). Mutually exclusive with `...`.

`more.args` [`list`]
A list of further arguments passed to `fun`. Default is an empty list.

`reg` [Registry]
Registry. If not explicitly passed, uses the default registry (see `setDefaultRegistry`).

Value

`data.table` with ids of added jobs stored in column "job.id".

See Also

`batchReduce`

Examples

```
# example using "..." and more.args
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
f = function(x, y) x^2 + y
ids = batchMap(f, x = 1:10, more.args = list(y = 100), reg = tmp)
getJobPars(reg = tmp)
testJob(6, reg = tmp) # 100 + 6^2 = 136

# vector recycling
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
f = function(...) list(...)
ids = batchMap(f, x = 1:3, y = 1:6, reg = tmp)
getJobPars(reg = tmp)

# example for an expand.grid()-like operation on parameters
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
ids = batchMap(paste, args = CJ(x = letters[1:3], y = 1:3), reg = tmp)
getJobPars(reg = tmp)
testJob(6, reg = tmp)
```

batchMapResults *Map Over Results to Create New Jobs*

Description

This function allows you to create new computational jobs (just like batchMap based on the results of a Registry.

Usage

```
batchMapResults(fun, ids = NULL, ..., more.args = list(), target,
  source = getDefaultRegistry())
```

Arguments

fun	[function] Function which takes the result as first (unnamed) argument.
ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of findDone. Invalid ids are ignored.
...	[ANY] Arguments to vectorize over (list or vector). Passed to batchMap.
more.args	[list] A list of further arguments passed to fun. Default is an empty list.
target	[Registry] Empty Registry where new jobs are created for.
source	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with ids of jobs added to target.

Note

The URI to the result files in registry source is hard coded as parameter in the target registry. This means that target is currently not portable between systems for computation.

See Also

Other Results: loadResult, reduceResultsList, reduceResults

Examples

```
# Source registry: calculate square of some numbers
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(function(x) list(square = x^2), x = 1:10, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)

# Target registry: calculate the square root on results of first registry
target = makeRegistry(file.dir = NA, make.default = FALSE)
batchMapResults(fun = function(x, y) list(sqrt = sqrt(x$square)), ids = 4:8,
  target = target, source = tmp)
submitJobs(reg = target)
waitForJobs(reg = target)

# Map old to new ids. First, get a table with results and parameters
results = unwrap(rjoin(getJobPars(reg = target), reduceResultsDataTable(reg = target)))
print(results)

# Parameter '.id' points to job.id in 'source'. Use a inner join to combine:
ijoin(results, unwrap(reduceResultsDataTable(reg = tmp)), by = c(".id" = "job.id"))
```

---

batchReduce *Reduce Operation for Batch Systems*

Description

A parallel and asynchronous Reduce for batch systems. Note that this function only defines the computational jobs. Each job reduces a certain number of elements on one slave. The actual computation is started with submitJobs. Results and partial results can be collected with reduceResultsList, reduceResults or loadResult.

Usage

```
batchReduce(fun, xs, init = NULL, chunks = seq_along(xs),
  more.args = list(), reg = getDefaultRegistry())
```

Arguments

fun	[function(aggr, x, ...)] Function to reduce xs with.
xs	[vector] Vector to reduce.
init	[ANY] Initial object for reducing. See Reduce.
chunks	[integer(length(xs))] Group for each element of xs. Can be generated with chunk.

more.args [list]
A list of additional arguments passed to fun.

reg [Registry]
Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with ids of added jobs stored in column "job.id".

See Also

batchMap

Examples

```
# define function to reduce on slave, we want to sum a vector
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
xs = 1:100
f = function(aggr, x) aggr + x

# sum 20 numbers on each slave process, i.e. 5 jobs
chunks = chunk(xs, chunk.size = 5)
batchReduce(fun = f, 1:100, init = 0, chunks = chunks, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)

# now reduce one final time on master
reduceResults(fun = function(aggr, job, res) f(aggr, res), reg = tmp)
```

---

btlapply *Synchronous Apply Functions*

Description

This is a set of functions acting as counterparts to the sequential popular apply functions in base R: btlapply for lapply and btmapply for mapply.

Internally, jobs are created using batchMap on the provided registry. If no registry is provided, a temporary registry (see argument file.dir of makeRegistry) and batchMap will be used. After all jobs are terminated (see waitForJobs), the results are collected and returned as a list.

Note that these functions are one suitable for short and fail-safe operations on batch system. If some jobs fail, you have to retrieve partial results from the registry directory yourself.

Usage

```
btlapply(X, fun, ..., resources = list(), n.chunks = NULL,
  chunk.size = NULL, reg = makeRegistry(file.dir = NA))

btmapply(fun, ..., more.args = list(), simplify = FALSE,
  use.names = TRUE, resources = list(), n.chunks = NULL,
  chunk.size = NULL, reg = makeRegistry(file.dir = NA))
```

Arguments

X	[vector] Vector to apply over.
fun	[function] Function to apply.
...	[ANY] Additional arguments passed to fun (btlapply) or vectors to map over (btmapply).
resources	[named list] Computational resources for the jobs to submit. The actual elements of this list (e.g. something like "walltime" or "nodes") depend on your template file, exceptions are outlined in the section 'Resources'. Default settings for a system can be set in the configuration file by defining the named list default.resources. Note that these settings are merged by name, e.g. merging list(walltime = 300) into list(walltime = 400, memory = 512) will result in list(walltime = 300, memory = 512). Same holds for individual job resources passed as additional column of ids (c.f. section 'Resources').
n.chunks	[integer(1)] Passed to chunk before submitJobs.
chunk.size	[integer(1)] Passed to chunk before submitJobs.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).
more.args	[list] Additional arguments passed to fun.
simplify	[logical(1)] Simplify the results using simplify2array?
use.names	[logical(1)] Use names of the input to name the output?

Value

list List with the results of the function call.

Examples

```
btlapply(1:3, function(x) x^2)
btmapply(function(x, y, z) x + y + z, x = 1:3, y = 1:3, more.args = list(z = 1), simplify = TRUE)
```

cfBrewTemplate *Cluster Functions Helper to Write Job Description Files*

Description

This function is only intended for use in your own cluster functions implementation.

Calls brew silently on your template, any error will lead to an exception. The file is stored at the same place as the corresponding job file in the "jobs"-subdir of your files directory.

Usage

```
cfBrewTemplate(reg, text, jc)
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).
text	[character(1)] String ready to be brewed. See cfReadBrewTemplate to read a template from the file system.
jc	[JobCollection)] Will be used as environment to brew the template file in. See JobCollection for a list of all available variables.

Value

character(1) . File path to brewed template file.

See Also

Other ClusterFunctionsHelper: cfHandleUnknownSubmitError, cfKillJob, cfReadBrewTemplate, makeClusterFunctions, makeSubmitJobResult, runOSCommand

cfHandleUnknownSubmitError *Cluster Functions Helper to Handle Unknown Errors*

Description

This function is only intended for use in your own cluster functions implementation.

Simply constructs a SubmitJobResult object with status code 101, NA as batch id and an informative error message containing the output of the OS command in output.

Usage

```
cfHandleUnknownSubmitError(cmd, exit.code, output)
```

Arguments

cmd	[character(1)] OS command used to submit the job, e.g. qsub.
exit.code	[integer(1)] Exit code of the OS command, should not be 0.
output	[character] Output of the OS command, hopefully an informative error message. If these are multiple lines in a vector, they are automatically joined.

Value

SubmitJobResult .

See Also

Other ClusterFunctionsHelper: cfBrewTemplate, cfKillJob, cfReadBrewTemplate, makeClusterFunctions, makeSubmitJobResult, runOSCommand

---

cfKillJob *Cluster Functions Helper to Kill Batch Jobs*

Description

This function is only intended for use in your own cluster functions implementation.

Calls the OS command to kill a job via system like this: "cmd batch.job.id". If the command returns an exit code > 0, the command is repeated after a 1 second sleep max.tries-1 times. If the command failed in all tries, an error is generated.

Usage

```
cfKillJob(reg, cmd, args = character(0L), max.tries = 3L,
  nodename = "localhost")
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).
cmd	[character(1)] OS command, e.g. "qdel".
args	[character] Arguments to cmd, including the batch id.

max.tries [integer(1)]
Number of total times to try execute the OS command in cases of failures. Default is 3.

nodename [character(1)]
Name of the SSH node to run the command on. If set to "localhost" (default), the command is not piped through SSH.

Value

TRUE on success. An exception is raised otherwise.

See Also

Other ClusterFunctionsHelper: cfBrewTemplate, cfHandleUnknownSubmitError, cfReadBrewTemplate, makeClusterFunctions, makeSubmitJobResult, runOSCommand

---

cfReadBrewTemplate *Cluster Functions Helper to Parse a Brew Template*

Description

This function is only intended for use in your own cluster functions implementation.

This function is only intended for use in your own cluster functions implementation. Simply reads your template file and returns it as a character vector.

Usage

```
cfReadBrewTemplate(template, comment.string = NA_character_)
```

Arguments

template [character(1)]
Path to template file which is then passed to brew.

comment.string [character(1)]
Ignore lines starting with this string.

Value

character .

See Also

Other ClusterFunctionsHelper: cfBrewTemplate, cfHandleUnknownSubmitError, cfKillJob, makeClusterFunctions, makeSubmitJobResult, runOSCommand

chunk *Chunk Jobs for Sequential Execution*

Description

Jobs can be partitioned into "chunks" to be executed sequentially on the computational nodes. Chunks are defined by providing a data frame with columns "job.id" and "chunk" (integer) to `submitJobs`. All jobs with the same chunk number will be grouped together on one node to form a single computational job.

The function `chunk` simply splits `x` into either a fixed number of groups, or into a variable number of groups with a fixed number of maximum elements.

The function `lpt` also groups `x` into a fixed number of chunks, but uses the actual values of `x` in a greedy "Longest Processing Time" algorithm. As a result, the maximum sum of elements in minimized.

`binpack` splits `x` into a variable number of groups whose sum of elements do not exceed the upper limit provided by `chunk.size`.

See examples of `estimateRuntimes` for an application of `binpack` and `lpt`.

Usage

```
chunk(x, n.chunks = NULL, chunk.size = NULL, shuffle = TRUE)

lpt(x, n.chunks = 1L)

binpack(x, chunk.size = max(x))
```

Arguments

`x`	[`numeric`] For `chunk` an atomic vector (usually the `job.id`). For `binpack` and `lpt`, the weights to group.
`n.chunks`	[`integer(1)`] Requested number of chunks. The function `chunk` distributes the number of elements in `x` evenly while `lpt` tries to even out the sum of elements in each chunk. If more chunks than necessary are requested, empty chunks are ignored. Mutually exclusive with `chunks.size`.
`chunk.size`	[`integer(1)`] Requested chunk size for each single chunk. For `chunk` this is the number of elements in `x`, for `binpack` the size is determined by the sum of values in `x`. Mutually exclusive with `n.chunks`.
`shuffle`	[`logical(1)`] Shuffles the groups. Default is `TRUE`.

Value

`integer` giving the chunk number for each element of `x`.

See Also

estimateRuntimes

Examples

```
ch = chunk(1:10, n.chunks = 2)
table(ch)

ch = chunk(rep(1, 10), chunk.size = 2)
table(ch)

set.seed(1)
x = runif(10)
ch = lpt(x, n.chunks = 2)
sapply(split(x, ch), sum)

set.seed(1)
x = runif(10)
ch = binpack(x, 1)
sapply(split(x, ch), sum)

# Job chunking
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
ids = batchMap(identity, 1:25, reg = tmp)

### Group into chunks with 10 jobs each
ids[, chunk := chunk(job.id, chunk.size = 10)]
print(ids[, .N, by = chunk])

### Group into 4 chunks
ids[, chunk := chunk(job.id, n.chunks = 4)]
print(ids[, .N, by = chunk])

### Submit to batch system
submitJobs(ids = ids, reg = tmp)

# Grouped chunking
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE)
prob = addProblem(reg = tmp, "prob1", data = iris, fun = function(job, data) nrow(data))
prob = addProblem(reg = tmp, "prob2", data = Titanic, fun = function(job, data) nrow(data))
algo = addAlgorithm(reg = tmp, "algo", fun = function(job, data, instance, i, ...) problem)
prob.designs = list(prob1 = data.table(), prob2 = data.table(x = 1:2))
algo.designs = list(algo = data.table(i = 1:3))
addExperiments(prob.designs, algo.designs, repls = 3, reg = tmp)

### Group into chunks of 5 jobs, but do not put multiple problems into the same chunk
# -> only one problem has to be loaded per chunk, and only once because it is cached
ids = getJobTable(reg = tmp)[, .(job.id, problem, algorithm)]
ids[, chunk := chunk(job.id, chunk.size = 5), by = "problem"]
ids[, chunk := .GRP, by = c("problem", "chunk")]
dcast(ids, chunk ~ problem)
```

clearRegistry *Remove All Jobs*

Description

Removes all jobs from a registry and calls sweepRegistry.

Usage

```
clearRegistry(reg = getDefaultRegistry())
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

See Also

Other Registry: getDefaultRegistry, loadRegistry, makeRegistry, removeRegistry, saveRegistry, sweepRegistry, syncRegistry

doJobCollection *Execute Jobs of a JobCollection*

Description

Executes every job in a JobCollection. This function is intended to be called on the slave.

Usage

```
doJobCollection(jc, output = NULL)
```

Arguments

jc	[JobCollection] Either an object of class "JobCollection" as returned by makeJobCollection or a string with the path to file containing a "JobCollection" as RDS file (as stored by submitJobs).
output	[character(1)] Path to a file to write the output to. Defaults to NULL which means that output is written to the active sink. Do not set this if your scheduler redirects output to a log file.

Value

character(1) : Hash of the JobCollection executed.

See Also

Other JobCollection: makeJobCollection

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(identity, 1:2, reg = tmp)
jc = makeJobCollection(1:2, reg = tmp)
doJobCollection(jc)
```

---

estimateRuntimes *Estimate Remaining Runtimes*

Description

Estimates the runtimes of jobs using the random forest implemented in **ranger**. Observed runtimes are retrieved from the Registry and runtimes are predicted for unfinished jobs.

The estimated remaining time is calculated in the print method. You may also pass n here to determine the number of parallel jobs which is then used in a simple Longest Processing Time (LPT) algorithm to give an estimate for the parallel runtime.

Usage

```
estimateRuntimes(tab, ..., reg = getDefaultRegistry())

## S3 method for class 'RuntimeEstimate'
print(x, n = 1L, ...)
```

Arguments

tab	[data.table] Table with column "job.id" and additional columns to predict the runtime. Observed runtimes will be looked up in the registry and serve as dependent variable. All columns in tab except "job.id" will be passed to ranger as independent variables to fit the model.
...	[ANY] Additional parameters passed to ranger. Ignored for the print method.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).
x	[RuntimeEstimate] Object to print.
n	[integer(1)] Number of parallel jobs to assume for runtime estimation.

Value

RuntimeEstimate which is a list with two named elements: "runtimes" is a data.table with columns "job.id", "runtime" (in seconds) and "type" ("estimated" if runtime is estimated, "observed" if runtime was observed). The other element of the list named "model"] contains the fitted random forest object.

See Also

binpack and lpt to chunk jobs according to their estimated runtimes.

Examples

```
# Create a simple toy registry
set.seed(1)
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE, seed = 1)
addProblem(name = "iris", data = iris, fun = function(data, ...) nrow(data), reg = tmp)
addAlgorithm(name = "nrow", function(instance, ...) nrow(instance), reg = tmp)
addAlgorithm(name = "ncol", function(instance, ...) ncol(instance), reg = tmp)
addExperiments(algo.designs = list(nrow = CJ(x = 1:50, y = letters[1:5])), reg = tmp)
addExperiments(algo.designs = list(ncol = CJ(x = 1:50, y = letters[1:5])), reg = tmp)

# We use the job parameters to predict runtimes
tab = unwrap(getJobPars(reg = tmp))

# First we need to submit some jobs so that the forest can train on some data.
# Thus, we just sample some jobs from the registry while grouping by factor variables.
ids = tab[, .SD[sample(nrow(.SD), 5)], by = c("problem", "algorithm", "y")]
setkeyv(ids, "job.id")
submitJobs(ids, reg = tmp)
waitForJobs(reg = tmp)

# We "simulate" some more realistic runtimes here to demonstrate the functionality:
# - Algorithm "ncol" is 5 times more expensive than "nrow"
# - x has no effect on the runtime
# - If y is "a" or "b", the runtimes are really high
runtime = function(algorithm, x, y) {
  ifelse(algorithm == "nrow", 100L, 500L) + 1000L * (y %in% letters[1:2])
}
tmp$status[ids, done := done + tab[ids, runtime(algorithm, x, y)]]
rjoin(sjoin(tab, ids), getJobStatus(ids, reg = tmp)[, c("job.id", "time.running")])

# Estimate runtimes:
est = estimateRuntimes(tab, reg = tmp)
print(est)
rjoin(tab, est$runtimes)
print(est, n = 10)

# Submit jobs with longest runtime first:
ids = est$runtimes[type == "estimated"][order(runtime, decreasing = TRUE)]
print(ids)
## Not run:
```

```
submitJobs(ids, reg = tmp)

## End(Not run)

# Group jobs into chunks with runtime < 1h
ids = est$runtimes[type == "estimated"]
ids[, chunk := binpack(runtime, 3600)]
print(ids)
print(ids[, list(runtime = sum(runtime)), by = chunk])
## Not run:
submitJobs(ids, reg = tmp)

## End(Not run)

# Group jobs into 10 chunks with similar runtime
ids = est$runtimes[type == "estimated"]
ids[, chunk := lpt(runtime, 10)]
print(ids[, list(runtime = sum(runtime)), by = chunk])
```

---

execJob *Execute a Single Jobs*

Description

Executes a single job (as created by makeJob) and returns its result. Also works for Experiments.

Usage

```
execJob(job)
```

Arguments

job	[Job	Experiment] Job/Experiment to execute.

Value

Result of the job.

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(identity, 1:2, reg = tmp)
job = makeJob(1, reg = tmp)
execJob(job)
```

findJobs *Find and Filter Jobs*

Description

These functions are used to find and filter jobs, depending on either their parameters (findJobs and findExperiments), their tags (findTagged), or their computational status (all other functions, see getStatus for an overview).

Note that findQueued, findRunning, findOnSystem and findExpired are somewhat heuristic and may report misleading results, depending on the state of the system and the ClusterFunctions implementation.

See JoinTables for convenient set operations (unions, intersects, differences) on tables with job ids.

Usage

```
findJobs(expr, ids = NULL, reg = getDefaultRegistry())

findExperiments(ids = NULL, prob.name = NA_character_,
  prob.pattern = NA_character_, algo.name = NA_character_,
  algo.pattern = NA_character_, prob.pars, algo.pars, repls = NULL,
  reg = getDefaultRegistry())

findSubmitted(ids = NULL, reg = getDefaultRegistry())

findNotSubmitted(ids = NULL, reg = getDefaultRegistry())

findStarted(ids = NULL, reg = getDefaultRegistry())

findNotStarted(ids = NULL, reg = getDefaultRegistry())

findDone(ids = NULL, reg = getDefaultRegistry())

findNotDone(ids = NULL, reg = getDefaultRegistry())

findErrors(ids = NULL, reg = getDefaultRegistry())

findOnSystem(ids = NULL, reg = getDefaultRegistry())

findRunning(ids = NULL, reg = getDefaultRegistry())

findQueued(ids = NULL, reg = getDefaultRegistry())

findExpired(ids = NULL, reg = getDefaultRegistry())

findTagged(tags = character(0L), ids = NULL,
  reg = getDefaultRegistry())
```

Arguments

expr	[expression] Predicate expression evaluated in the job parameters. Jobs for which expr evaluates to TRUE are returned.
ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).
prob.name	[character] Exact name of the problem (no substring matching). If not provided, all problems are matched.
prob.pattern	[character] Regular expression pattern to match problem names. If not provided, all problems are matched.
algo.name	[character] Exact name of the problem (no substring matching). If not provided, all algorithms are matched.
algo.pattern	[character] Regular expression pattern to match algorithm names. If not provided, all algorithms are matched.
prob.pars	[expression] Predicate expression evaluated in the problem parameters.
algo.pars	[expression] Predicate expression evaluated in the algorithm parameters.
repls	[integer] Whitelist of replication numbers. If not provided, all replications are matched.
tags	[character] Return jobs which are tagged with any of the tags provided.

Value

data.table with column "job.id" containing matched jobs.

See Also

getStatus JoinTables

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(identity, i = 1:3, reg = tmp)
ids = findNotSubmitted(reg = tmp)
```

```
# get all jobs:
findJobs(reg = tmp)

# filter for jobs with parameter i >= 2
findJobs(i >= 2, reg = tmp)

# filter on the computational status
findSubmitted(reg = tmp)
findNotDone(reg = tmp)

# filter on tags
addJobTags(2:3, "my_tag", reg = tmp)
findTagged(tags = "my_tag", reg = tmp)

# combine filter functions using joins
# -> jobs which are not done and not tagged (using an anti-join):
ajoin(findNotDone(reg = tmp), findTagged("my_tag", reg = tmp))
```

`getDefaultRegistry` *Get and Set the Default Registry*

Description

`getDefaultRegistry` returns the registry currently set as default (or stops with an exception if none is set). `setDefaultRegistry` sets a registry as default.

Usage

```
getDefaultRegistry()

setDefaultRegistry(reg)
```

Arguments

`reg`	[`Registry`] Registry. If not explicitly passed, uses the default registry (see `setDefaultRegistry`).

See Also

Other Registry: `clearRegistry`, `loadRegistry`, `makeRegistry`, `removeRegistry`, `saveRegistry`, `sweepRegistry`, `syncRegistry`

getErrorMessages *Retrieve Error Messages*

Description

Extracts error messages from the internal data base and returns them in a table.

Usage

```
getErrorMessages(ids = NULL, missing.as.error = FALSE,
  reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of findErrors. Invalid ids are ignored.
missing.as.error	[logical(1)] Treat missing results as errors? If TRUE, the error message "[not terminated]" is imputed for jobs which have not terminated. Default is FALSE
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with columns "job.id", "terminated" (logical), "error" (logical) and "message" (string).

See Also

Other debug: getStatus, grepLogs, killJobs, resetJobs, showLog, testJob

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
fun = function(i) if (i == 3) stop(i) else i
ids = batchMap(fun, i = 1:5, reg = tmp)
submitJobs(1:4, reg = tmp)
waitForJobs(1:4, reg = tmp)
getErrorMessages(ids, reg = tmp)
getErrorMessages(ids, missing.as.error = TRUE, reg = tmp)
```

getJobTable *Query Job Information*

Description

getJobStatus returns the internal table which stores information about the computational status of jobs, getJobPars a table with the job parameters, getJobResources a table with the resources which were set to submit the jobs, and getJobTags the tags of the jobs (see Tags).

getJobTable returns all these tables joined.

Usage

```
getJobTable(ids = NULL, reg = getDefaultRegistry())

getJobStatus(ids = NULL, reg = getDefaultRegistry())

getJobResources(ids = NULL, reg = getDefaultRegistry())

getJobPars(ids = NULL, reg = getDefaultRegistry())

getJobTags(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with the following columns (not necessarily in this order):

job.id Unique Job ID as integer.

submitted Time the job was submitted to the batch system as POSIXct.

started Time the job was started on the batch system as POSIXct.

done Time the job terminated (successfully or with an error) as POSIXct.

error Either NA if the job terminated successfully or the error message.

mem.used Estimate of the memory usage.

batch.id Batch ID as reported by the scheduler.

log.file Log file. If missing, defaults to [job.hash].log.

job.hash Unique string identifying the job or chunk.

time.queued Time in seconds (as difftime) the job was queued.

time.running Time in seconds (as difftime) the job was running.

pars List of parameters/arguments for this job.

resources List of computational resources set for this job.

tags Tags as joined string, delimited by ",".

problem Only for ExperimentRegistry: the problem identifier.

algorithm Only for ExperimentRegistry: the algorithm identifier.

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
f = function(x) if (x < 0) stop("x must be > 0") else sqrt(x)
batchMap(f, x = c(-1, 0, 1), reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)
addJobTags(1:2, "tag1", reg = tmp)
addJobTags(2, "tag2", reg = tmp)

# Complete table:
getJobTable(reg = tmp)

# Job parameters:
getJobPars(reg = tmp)

# Set and retrieve tags:
getJobTags(reg = tmp)

# Job parameters with tags right-joined:
rjoin(getJobPars(reg = tmp), getJobTags(reg = tmp))
```

---

getStatus *Summarize the Computational Status*

Description

This function gives an encompassing overview over the computational status on your system. The status can be one or many of the following:

- "defined": Jobs which are defined via batchMap or addExperiments, but are not yet submitted.
- "submitted": Jobs which are submitted to the batch system via submitJobs, scheduled for execution.
- "started": Jobs which have been started.
- "done": Jobs which terminated successfully.
- "error": Jobs which terminated with an exception.

- "running": Jobs which are listed by the cluster functions to be running on the live system. Not supported for all cluster functions.
- "queued": Jobs which are listed by the cluster functions to be queued on the live system. Not supported for all cluster functions.
- "system": Jobs which are listed by the cluster functions to be queued or running. Not supported for all cluster functions.
- "expired": Jobs which have been submitted, but vanished from the live system. Note that this is determined heuristically and may include some false positives.

Here, a job which terminated successfully counts towards the jobs which are submitted, started and done. To retrieve the corresponding job ids, see findJobs.

Usage

```
getStatus(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table (with class "Status" for printing).

See Also

findJobs

Other debug: getErrorMessages, grepLogs, killJobs, resetJobs, showLog, testJob

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
fun = function(i) if (i == 3) stop(i) else i
ids = batchMap(fun, i = 1:5, reg = tmp)
submitJobs(ids = 1:4, reg = tmp)
waitForJobs(reg = tmp)

tab = getStatus(reg = tmp)
print(tab)
str(tab)
```

grepLogs *Grep Log Files for a Pattern*

Description

Crawls through log files and reports jobs with lines matching the `pattern`. See `showLog` for an example.

Usage

```
grepLogs(ids = NULL, pattern, ignore.case = FALSE, fixed = FALSE,
  reg = getDefaultRegistry())
```

Arguments

`ids`	[`data.frame` or `integer`] A `data.frame` (or `data.table`) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of `findStarted`. Invalid ids are ignored.
`pattern`	[`character(1L)`] Regular expression or string (see `fixed`).
`ignore.case`	[`logical(1L)`] If `TRUE` the match will be performed case insensitively.
`fixed`	[`logical(1L)`] If `FALSE` (default), `pattern` is a regular expression and a fixed string otherwise.
`reg`	[`Registry`] Registry. If not explicitly passed, uses the default registry (see `setDefaultRegistry`).

Value

`data.table` with columns "job.id" and "message".

See Also

Other debug: `getErrorMessages`, `getStatus`, `killJobs`, `resetJobs`, `showLog`, `testJob`

JobNames *Set and Retrieve Job Names*

Description

Set custom names for jobs. These are passed to the template as 'job.name'. If no custom name is set (or any of the job names of the chunk is missing), the job hash is used as job name. Individual job names can be accessed via `jobs$job.name`.

Usage

```
setJobNames(ids = NULL, names, reg = getDefaultRegistry())

getJobNames(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
names	[character] Character vector of the same length as provided ids.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

`setJobNames` returns `NULL` invisibly, `getJobTable` returns a `data.table` with columns `job.id` and `job.name`.

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
ids = batchMap(identity, 1:10, reg = tmp)
setJobNames(ids, letters[1:nrow(ids)], reg = tmp)
getJobNames(reg = tmp)
```

JoinTables *Inner, Left, Right, Outer, Semi and Anti Join for Data Tables*

Description

These helper functions perform join operations on data tables. Most of them are basically one-liners. See http://rpubs.com/ronasta/join_data_tables for a overview of join operations in data table or alternatively **dplyr**'s vignette on two table verbs.

Usage

```
ijoin(x, y, by = NULL)

ljoin(x, y, by = NULL)

rjoin(x, y, by = NULL)

ojoin(x, y, by = NULL)

sjoin(x, y, by = NULL)

ajoin(x, y, by = NULL)

ujoin(x, y, all.y = FALSE, by = NULL)
```

Arguments

x	[data.frame] First data.frame to join.
y	[data.frame] Second data.frame to join.
by	[character] Column name(s) of variables used to match rows in x and y. If not provided, a heuristic similar to the one described in the **dplyr** vignette is used: 1. If x is keyed, the existing key will be used if y has the same column(s). 2. If x is not keyed, the intersect of common columns names is used if not empty. 3. Raise an exception. You may pass a named character vector to merge on columns with different names in x and y: `by = c("x.id" = "y.id")` will match x's "x.id" column with y\'s "y.id" column.
all.y	[logical(1)] Keep columns of y which are not in x?

Value

data.table with key identical to by.

Examples

```
# Create two tables for demonstration
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(identity, x = 1:6, reg = tmp)
x = getJobPars(reg = tmp)
y = findJobs(x >= 2 & x <= 5, reg = tmp)
y$extra.col = head(letters, nrow(y))

# Inner join: similar to intersect(): keep all columns of x and y with common matches
ijoin(x, y)

# Left join: use all ids from x, keep all columns of x and y
ljoin(x, y)

# Right join: use all ids from y, keep all columns of x and y
rjoin(x, y)

# Outer join: similar to union(): keep all columns of x and y with matches in x or y
ojoin(x, y)

# Semi join: filter x with matches in y
sjoin(x, y)

# Anti join: filter x with matches not in y
ajoin(x, y)

# Updating join: Replace values in x with values in y
ujoin(x, y)
```

---

killJobs *Kill Jobs*

Description

Kill jobs which are currently running on the batch system.

In case of an error when killing, the function tries - after a short sleep - to kill the remaining batch jobs again. If this fails three times for some jobs, the function gives up. Jobs that could be successfully killed are reset in the Registry.

Usage

```
killJobs(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of findOnSystem. Invalid ids are ignored.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with columns "job.id", the corresponding "batch.id" and the logical flag "killed" indicating success.

See Also

Other debug: getErrorMessages, getStatus, grepLogs, resetJobs, showLog, testJob

---

loadRegistry *Load a Registry from the File System*

Description

Loads a registry from its file.dir.

Multiple R sessions accessing the same registry simultaneously can lead to database inconsistencies. This is especially dangerous if the same file.dir is accessed from multiple machines, e.g. via a mount.

If you just need to check on the status or peek into some preliminary results while another process is still submitting or waiting for pending results, you can load the registry in a read-only mode. All operations that need to change the registry will raise an exception in this mode. Files communicated back by the computational nodes are parsed to update the registry in memory while the registry on the file system remains unchanged.

A heuristic tries to detect if the registry has been altered in the background by an other process and in this case automatically restricts the current registry to read-only mode. However, you should rely on this heuristic to work flawlessly. Thus, set to writeable to TRUE if and only if you are absolutely sure that other state-changing processes are terminated.

If you need write access, load the registry with writeable set to TRUE.

Usage

```
loadRegistry(file.dir, work.dir = NULL, conf.file = findConfFile(),
  make.default = TRUE, writeable = FALSE)
```

Arguments

`file.dir` [character(1)]
Path where all files of the registry are saved. Default is directory "registry" in the current working directory. The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well.

If you pass NA, a temporary directory will be used. This way, you can create disposable registries for btlapply or examples. By default, the temporary directory tempdir() will be used. If you want to use another directory, e.g. a directory which is shared between nodes, you can set it in your configuration file by setting the variable temp.dir.

`work.dir` [character(1)]
Working directory for R process for running jobs. Defaults to the working directory currently set during Registry construction (see getwd). loadRegistry uses the stored work.dir, but you may also explicitly overwrite it, e.g., after switching to another system.

The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well.

`conf.file` [character(1)]
Path to a configuration file which is sourced while the registry is created. In the configuration file you can define how **batchtools** interacts with the system via ClusterFunctions. Separating the configuration of the underlying host system from the R code allows to easily move computation to another site.

The file lookup is implemented in the internal (but exported) function findConfFile which returns the first file found of the following candidates:

1. File "batchtools.conf.R" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. File "batchtools.conf.R" in the current working directory.
3. File "config.R" in the user configuration directory as reported by rappdirs::user_config_dir("ba (depending on OS, e.g., on linux this usually resolves to "~/.config/batchtools/config.R").
4. ".batchtools.conf.R" in the home directory ("~").
5. "config.R" in the site config directory as reported by rappdirs::site_config_dir("batchtools") (depending on OS). This file can be used for admins to set sane defaults for a computation site.

Set to NA if you want to suppress reading any configuration file. If a configuration file is found, it gets sourced inside the environment of the registry after the defaults for all variables are set. Therefore you can set and overwrite slots, e.g. default.resources = list(walltime = 3600) to set default resources or "max.concurrent.jobs" to limit the number of jobs allowed to run simultaneously on the system.

`make.default` [logical(1)]
If set to TRUE, the created registry is saved inside the package namespace and acts as default registry. You might want to switch this off if you work with multiple registries simultaneously. Default is TRUE.

writeable [logical(1)]
Loads the registry in read-write mode. Default is FALSE.

Value

Registry .

See Also

Other Registry: clearRegistry, getDefaultRegistry, makeRegistry, removeRegistry, saveRegistry, sweepRegistry, syncRegistry

---

loadResult *Load the Result of a Single Job*

Description

Loads the result of a single job.

Usage

```
loadResult(id, reg = getDefaultRegistry())
```

Arguments

id	[integer(1) or data.table] Single integer to specify the job or a data.table with column job.id and exactly one row.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

ANY . The stored result.

See Also

Other Results: batchMapResults, reduceResultsList, reduceResults

makeClusterFunctions *ClusterFunctions Constructor*

Description

This is the constructor used to create *custom* cluster functions. Note that some standard implementations for TORQUE, Slurm, LSF, SGE, etc. ship with the package.

Usage

```
makeClusterFunctions(name, submitJob, killJob = NULL,
  listJobsQueued = NULL, listJobsRunning = NULL,
  array.var = NA_character_, store.job.collection = FALSE,
  store.job.files = FALSE, scheduler.latency = 0, fs.latency = 0,
  hooks = list())
```

Arguments

name	[character(1)] Name of cluster functions.
submitJob	[function(reg, jc, ...)] Function to submit new jobs. Must return a SubmitJobResult object. The arguments are reg (Registry) and jobs (JobCollection).
killJob	[function(reg, batch.id)] Function to kill a job on the batch system. Make sure that you definitely kill the job! Return value is currently ignored. Must have the arguments reg (Registry) and batch.id (character(1) as returned by submitJob). Note that there is a helper function cfKillJob to repeatedly try to kill jobs. Set killJob to NULL if killing jobs cannot be supported.
listJobsQueued	[function(reg)] List all queued jobs on the batch system for the current user. Must return an character vector of batch ids, same format as they are returned by submitJob. Set listJobsQueued to NULL if listing of queued jobs is not supported.
listJobsRunning	[function(reg)] List all running jobs on the batch system for the current user. Must return an character vector of batch ids, same format as they are returned by submitJob. It does not matter if you return a few job ids too many (e.g. all for the current user instead of all for the current registry), but you have to include all relevant ones. Must have the argument are reg (Registry). Set listJobsRunning to NULL if listing of running jobs is not supported.
array.var	[character(1)] Name of the environment variable set by the scheduler to identify IDs of job arrays. Default is NA for no array support.

store.job.collection
: [logical(1)]
Flag to indicate that the cluster function implementation of submitJob can not directly handle JobCollection objects. If set to FALSE, the JobCollection is serialized to the file system before submitting the job.

store.job.files
: [logical(1)]
Flag to indicate that job files need to be stored in the file directory. If set to FALSE (default), the job file is created in a temporary directory, otherwise (or if the debug mode is enabled) in the subdirectory jobs of the file.dir.

scheduler.latency
: [numeric(1)]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling submitJobs.

fs.latency
: [numeric(1)]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

hooks
: [list]
Named list of functions which will we called on certain events like "pre.submit" or "post.sync". See Hooks.

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlurm, makeClusterFunctionsSocket makeClusterFunctionsTORQUE

Other ClusterFunctionsHelper: cfBrewTemplate, cfHandleUnknownSubmitError, cfKillJob, cfReadBrewTemplate, makeSubmitJobResult, runOSCommand

---

makeClusterFunctionsDocker

ClusterFunctions for Docker

Description

Cluster functions for Docker/Docker Swarm (https://docs.docker.com/swarm/).

The submitJob function executes docker [docker.args] run --detach=true [image.args] [resources] [image] [c
Arguments docker.args, image.args and image can be set on construction. The resources part takes the named resources ncpus and memory from submitJobs and maps them to the arguments --cpu-shares and --memory (in Megabytes). The resource threads is mapped to the environment variables "OMP_NUM_THREADS" and "OPENBLAS_NUM_THREADS". To reliably identify jobs in the swarm, jobs are labeled with "batchtools=[job.hash]" and named using the current login name (label "user") and the job hash (label "batchtools").

listJobsRunning uses docker [docker.args] ps --format={{.ID}} to filter for running jobs.

killJobs uses docker [docker.args] kill [batch.id] to filter for running jobs.

These cluster functions use a Hook to remove finished jobs before a new submit and every time the Registry is synchronized (using syncRegistry). This is currently required because docker does not remove terminated containers automatically. Use docker ps -a --filter 'label=batchtools' --filter 'status=exi to identify and remove terminated containers manually (or usa a cron job).

Usage

```
makeClusterFunctionsDocker(image, docker.args = character(0L),
  image.args = character(0L), scheduler.latency = 1, fs.latency = 65)
```

Arguments

image	[character(1)] Name of the docker image to run.
docker.args	[character] Additional arguments passed to "docker" *before* the command ("run", "ps" or "kill") to execute (e.g., the docker host).
image.args	[character] Additional arguments passed to "docker run" (e.g., to define mounts or environment variables).
scheduler.latency	[numeric(1)] Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling submitJobs.
fs.latency	[numeric(1)] Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

See Also

Other ClusterFunctions: makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMult
makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlu
makeClusterFunctionsSocket, makeClusterFunctionsTORQUE, makeClusterFunctions

makeClusterFunctionsInteractive
ClusterFunctions for Sequential Execution in the Running R Session

Description

All jobs are executed sequentially using the current R process in which submitJobs is called. Thus, submitJob blocks the session until the job has finished. The main use of this ClusterFunctions implementation is to test and debug programs on a local computer.

Listing jobs returns an empty vector (as no jobs can be running when you call this) and killJob is not implemented for the same reasons.

Usage

```
makeClusterFunctionsInteractive(external = FALSE, write.logs = TRUE,
  fs.latency = 0)
```

Arguments

external	[logical(1)] If set to TRUE, jobs are started in a fresh R session instead of currently active but still waits for its termination. Default is FALSE.
write.logs	[logical(1)] Sink the output to log files. Turning logging off can increase the speed of calculations but makes it very difficult to debug. Default is TRUE.
fs.latency	[numeric(1)] Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsLSF, makeClusterFunctionsMulticore
makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlu
makeClusterFunctionsSocket, makeClusterFunctionsTORQUE, makeClusterFunctions

makeClusterFunctionsLSF *ClusterFunctions for LSF Systems*

Description

Cluster functions for LSF (https://www.ibm.com/us-en/marketplace/hpc-workload-management).

Job files are created based on the brew template `template.file`. This file is processed with brew and then submitted to the queue using the `bsub` command. Jobs are killed using the `bkill` command and the list of running jobs is retrieved using `bjobs -u $USER -w`. The user must have the appropriate privileges to submit, delete and list jobs on the cluster (this is usually the case).

The template file can access all resources passed to `submitJobs` as well as all variables stored in the `JobCollection`. It is the template file's job to choose a queue for the job and handle the desired resource allocations.

Usage

```
makeClusterFunctionsLSF(template = "lsf", scheduler.latency = 1,
  fs.latency = 65)
```

Arguments

`template` [character(1)]
Either a path to a **brew** template file (with extension "tmpl"), or a short descriptive name enabling the following heuristic for the file lookup:

1. "batchtools.[template].tmpl" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. "batchtools.[template].tmpl" in the current working directory.
3. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
4. ".batchtools.[template].tmpl" in the home directory.
5. "[template].tmpl" in the package installation directory in the subfolder "templates".

`scheduler.latency` [numeric(1)]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling `submitJobs`.

`fs.latency` [numeric(1)]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to `0` to disable the heuristic, e.g. if you are working on a local file system.

Value

`ClusterFunctions` .

Note

Array jobs are currently not supported.

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlurm, makeClusterFunctionsSocket, makeClusterFunctionsTOR makeClusterFunctions

---

makeClusterFunctionsMulticore

ClusterFunctions for Parallel Multicore Execution

Description

Jobs are spawned asynchronously using the functions `mcparallel` and `mccollect` (both in **parallel**). Does not work on Windows, use `makeClusterFunctionsSocket` instead.

Usage

```
makeClusterFunctionsMulticore(ncpus = NA_integer_, fs.latency = 0)
```

Arguments

ncpus	[integer(1)] Number of CPUs. Default is to use all logical cores. The total number of cores "available" can be set via the option `mc.cores` and defaults to the heuristic implemented in `detectCores`.
fs.latency	[numeric(1)] Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to `0` to disable the heuristic, e.g. if you are working on a local file system.

Value

`ClusterFunctions` .

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH makeClusterFunctionsSlurm, makeClusterFunctionsSocket, makeClusterFunctionsTORQUE, makeClusterFunctions

`makeClusterFunctionsOpenLava` *ClusterFunctions for OpenLava*

Description

Cluster functions for OpenLava.

Job files are created based on the brew template `template`. This file is processed with brew and then submitted to the queue using the `bsub` command. Jobs are killed using the `bkill` command and the list of running jobs is retrieved using `bjobs -u $USER -w`. The user must have the appropriate privileges to submit, delete and list jobs on the cluster (this is usually the case).

The template file can access all resources passed to `submitJobs` as well as all variables stored in the `JobCollection`. It is the template file's job to choose a queue for the job and handle the desired resource allocations.

Usage

```
makeClusterFunctionsOpenLava(template = "openlava",
  scheduler.latency = 1, fs.latency = 65)
```

Arguments

`template` [`character(1)`]
Either a path to a **brew** template file (with extension "tmpl"), or a short descriptive name enabling the following heuristic for the file lookup:

1. "batchtools.[template].tmpl" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. "batchtools.[template].tmpl" in the current working directory.
3. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
4. ".batchtools.[template].tmpl" in the home directory.
5. "[template].tmpl" in the package installation directory in the subfolder "templates".

`scheduler.latency` [`numeric(1)`]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling `submitJobs`.

`fs.latency` [`numeric(1)`]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to `0` to disable the heuristic, e.g. if you are working on a local file system.

Value

`ClusterFunctions` .

Note

Array jobs are currently not supported.

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsSGE, makeClusterFunctionsSS makeClusterFunctionsSlurm, makeClusterFunctionsSocket, makeClusterFunctionsTORQUE, makeClusterFunctions

---

`makeClusterFunctionsSGE`

ClusterFunctions for SGE Systems

Description

Cluster functions for Univa Grid Engine / Oracle Grid Engine / Sun Grid Engine (http://www.univa.com/).

Job files are created based on the brew template `template`. This file is processed with brew and then submitted to the queue using the `qsub` command. Jobs are killed using the `qdel` command and the list of running jobs is retrieved using `qselect`. The user must have the appropriate privileges to submit, delete and list jobs on the cluster (this is usually the case).

The template file can access all resources passed to `submitJobs` as well as all variables stored in the `JobCollection`. It is the template file's job to choose a queue for the job and handle the desired resource allocations.

Usage

```
makeClusterFunctionsSGE(template = "sge", nodename = "localhost",
  scheduler.latency = 1, fs.latency = 65)
```

Arguments

`template` `[character(1)]`
Either a path to a **brew** template file (with extension "tmpl"), or a short descriptive name enabling the following heuristic for the file lookup:

1. "batchtools.[template].tmpl" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. "batchtools.[template].tmpl" in the current working directory.
3. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
4. ".batchtools.[template].tmpl" in the home directory.
5. "[template].tmpl" in the package installation directory in the subfolder "templates".

nodename [character(1)]
Nodename of the master host. All commands are send via SSH to this host. Only works iff

1. Passwordless authentication (e.g., via SSH public key authentication) is set up.
2. The file directory is shared across machines, e.g. mounted via SSHFS.
3. Either the absolute path to the file.dir is identical on the machines, or paths are provided relative to the home directory. Symbolic links should work.

scheduler.latency
[numeric(1)]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling submitJobs.

fs.latency [numeric(1)]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

Note

Array jobs are currently not supported.

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSSH, makeClusterFunctionsSlurm, makeClusterFunctionsSocket, makeClusterFunctionsTOR makeClusterFunctions

---

makeClusterFunctionsSlurm

ClusterFunctions for Slurm Systems

Description

Cluster functions for Slurm (http://slurm.schedmd.com/).

Job files are created based on the brew template template.file. This file is processed with brew and then submitted to the queue using the sbatch command. Jobs are killed using the scancel command and the list of running jobs is retrieved using squeue. The user must have the appropriate privileges to submit, delete and list jobs on the cluster (this is usually the case).

The template file can access all resources passed to submitJobs as well as all variables stored in the JobCollection. It is the template file's job to choose a queue for the job and handle the desired resource allocations.

Note that you might have to specify the cluster name here if you do not want to use the default, otherwise the commands for listing and killing jobs will not work.

Usage

```
makeClusterFunctionsSlurm(template = "slurm", array.jobs = TRUE,
  nodename = "localhost", scheduler.latency = 1, fs.latency = 65)
```

Arguments

`template` [character(1)]
Either a path to a **brew** template file (with extension "tmpl"), or a short descriptive name enabling the following heuristic for the file lookup:

1. "batchtools.[template].tmpl" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. "batchtools.[template].tmpl" in the current working directory.
3. "[template].tmpl" in the user config directory (see user_config_dir); on linux this is usually "~/.config/batchtools/[template].tmpl".
4. ".batchtools.[template].tmpl" in the home directory.
5. "[template].tmpl" in the package installation directory in the subfolder "templates".

`array.jobs` [logical(1)]
If array jobs are disabled on the computing site, set to FALSE.

`nodename` [character(1)]
Nodename of the master host. All commands are send via SSH to this host. Only works iff

1. Passwordless authentication (e.g., via SSH public key authentication) is set up.
2. The file directory is shared across machines, e.g. mounted via SSHFS.
3. Either the absolute path to the file.dir is identical on the machines, or paths are provided relative to the home directory. Symbolic links should work.

`scheduler.latency` [numeric(1)]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling submitJobs.

`fs.latency` [numeric(1)]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSocket, makeClusterFunctionsTORQU makeClusterFunctions

---

makeClusterFunctionsSocket

ClusterFunctions for Parallel Socket Execution

Description

Jobs are spawned asynchronously using the package **snow**.

Usage

```
makeClusterFunctionsSocket(ncpus = NA_integer_, fs.latency = 65)
```

Arguments

ncpus	[integer(1)] Number of CPUs. Default is to use all logical cores. The total number of cores "available" can be set via the option mc.cores and defaults to the heuristic implemented in detectCores.
fs.latency	[numeric(1)] Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to 0 to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlurm, makeClusterFunctionsTORQUE makeClusterFunctions

makeClusterFunctionsSSH *ClusterFunctions for Remote SSH Execution*

Description

Jobs are spawned by starting multiple R sessions via `Rscript` over SSH. If the hostname of the `Worker` equals "localhost", `Rscript` is called directly so that you do not need to have an SSH client installed.

Usage

```
makeClusterFunctionsSSH(workers, fs.latency = 65)
```

Arguments

`workers`	[`list` of `Worker`] List of Workers as constructed with `Worker`.
`fs.latency`	[`numeric(1)`] Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to `0` to disable the heuristic, e.g. if you are working on a local file system.

Value

`ClusterFunctions` .

Note

If you use a custom ".ssh/config" file, make sure your ProxyCommand passes '-q' to ssh, otherwise each output will end with the message "Killed by signal 1" and this will break the communication with the nodes.

See Also

Other ClusterFunctions: `makeClusterFunctionsDocker`, `makeClusterFunctionsInteractive`, `makeClusterFunctionsLSF`, `makeClusterFunctionsMulticore`, `makeClusterFunctionsOpenLava`, `makeClusterFunctionsSGE`, `makeClusterFunctionsSlurm`, `makeClusterFunctionsSocket`, `makeClusterFunctionsTOR`
`makeClusterFunctions`

Examples

```
## Not run:
# cluster functions for multicore execution on the local machine
makeClusterFunctionsSSH(list(Worker$new("localhost", ncpus = 2)))

## End(Not run)
```

makeClusterFunctionsTORQUE *ClusterFunctions for OpenPBS/TORQUE Systems*

Description

Cluster functions for TORQUE/PBS (http://www.adaptivecomputing.com/products/open-source/torque/).

Job files are created based on the brew template `template.file`. This file is processed with brew and then submitted to the queue using the `qsub` command. Jobs are killed using the `qdel` command and the list of running jobs is retrieved using `qselect`. The user must have the appropriate privileges to submit, delete and list jobs on the cluster (this is usually the case).

The template file can access all resources passed to `submitJobs` as well as all variables stored in the `JobCollection`. It is the template file's job to choose a queue for the job and handle the desired resource allocations.

Usage

```
makeClusterFunctionsTORQUE(template = "torque", scheduler.latency = 1,
  fs.latency = 65)
```

Arguments

`template` [`character(1)`]
Either a path to a **brew** template file (with extension "tmpl"), or a short descriptive name enabling the following heuristic for the file lookup:

1. "batchtools.[template].tmpl" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. "batchtools.[template].tmpl" in the current working directory.
3. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
4. ".batchtools.[template].tmpl" in the home directory.
5. "[template].tmpl" in the package installation directory in the subfolder "templates".

`scheduler.latency` [`numeric(1)`]
Time to sleep after important interactions with the scheduler to ensure a sane state. Currently only triggered after calling `submitJobs`.

`fs.latency` [`numeric(1)`]
Expected maximum latency of the file system, in seconds. Set to a positive number for network file systems like NFS which enables more robust (but also more expensive) mechanisms to access files and directories. Usually safe to set to `0` to disable the heuristic, e.g. if you are working on a local file system.

Value

ClusterFunctions .

See Also

Other ClusterFunctions: makeClusterFunctionsDocker, makeClusterFunctionsInteractive, makeClusterFunctionsLSF, makeClusterFunctionsMulticore, makeClusterFunctionsOpenLava, makeClusterFunctionsSGE, makeClusterFunctionsSSH, makeClusterFunctionsSlurm, makeClusterFunctionsSocket makeClusterFunctions

---

makeExperimentRegistry

ExperimentRegistry Constructor

Description

makeExperimentRegistry constructs a special Registry which is suitable for the definition of large scale computer experiments.

Each experiments consists of a Problem and an Algorithm. These can be parametrized with addExperiments to actually define computational jobs.

Usage

```
makeExperimentRegistry(file.dir = "registry", work.dir = getwd(),
  conf.file = findConfFile(), packages = character(0L),
  namespaces = character(0L), source = character(0L),
  load = character(0L), seed = NULL, make.default = TRUE)
```

Arguments

file.dir	[character(1)] Path where all files of the registry are saved. Default is directory "registry" in the current working directory. The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well. If you pass NA, a temporary directory will be used. This way, you can create disposable registries for btlapply or examples. By default, the temporary directory tempdir() will be used. If you want to use another directory, e.g. a directory which is shared between nodes, you can set it in your configuration file by setting the variable temp.dir.
work.dir	[character(1)] Working directory for R process for running jobs. Defaults to the working directory currently set during Registry construction (see getwd). loadRegistry uses the stored work.dir, but you may also explicitly overwrite it, e.g., after switching to another system.

The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well.

`conf.file` [character(1)]
Path to a configuration file which is sourced while the registry is created. In the configuration file you can define how **batchtools** interacts with the system via ClusterFunctions. Separating the configuration of the underlying host system from the R code allows to easily move computation to another site.

The file lookup is implemented in the internal (but exported) function `findConfFile` which returns the first file found of the following candidates:

1. File "batchtools.conf.R" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH".
2. File "batchtools.conf.R" in the current working directory.
3. File "config.R" in the user configuration directory as reported by `rappdirs::user_config_dir("ba`
(depending on OS, e.g., on linux this usually resolves to "~/.config/batchtools/config.R").
4. ".batchtools.conf.R" in the home directory ("~").
5. "config.R" in the site config directory as reported by `rappdirs::site_config_dir("batchtools")`
(depending on OS). This file can be used for admins to set sane defaults for a computation site.

Set to NA if you want to suppress reading any configuration file. If a configuration file is found, it gets sourced inside the environment of the registry after the defaults for all variables are set. Therefore you can set and overwrite slots, e.g. `default.resources = list(walltime = 3600)` to set default resources or "max.concurrent.jobs" to limit the number of jobs allowed to run simultaneously on the system.

`packages` [character]
Packages that will always be loaded on each node. Uses require internally. Default is `character(0)`.

`namespaces` [character]
Same as packages, but the packages will not be attached. Uses requireNamespace internally. Default is `character(0)`.

`source` [character]
Files which should be sourced on the slaves prior to executing a job. Calls sys.source using the .GlobalEnv.

`load` [character]
Files which should be loaded on the slaves prior to executing a job. Calls load using the .GlobalEnv.

`seed` [integer(1)]
Start seed for jobs. Each job uses the (seed + job.id) as seed. Default is a random number in the range [1, `.Machine$integer.max/2`].

`make.default` [logical(1)]
If set to TRUE, the created registry is saved inside the package namespace and acts as default registry. You might want to switch this off if you work with multiple registries simultaneously. Default is TRUE.

Value

ExperimentRegistry .

Examples

```
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE)

# Definde one problem, two algorithms and add them with some parameters:
addProblem(reg = tmp, "p1",
  fun = function(job, data, n, mean, sd, ...) rnorm(n, mean = mean, sd = sd))
addAlgorithm(reg = tmp, "a1", fun = function(job, data, instance, ...) mean(instance))
addAlgorithm(reg = tmp, "a2", fun = function(job, data, instance, ...) median(instance))
ids = addExperiments(reg = tmp, list(p1 = CJ(n = c(50, 100), mean = -2:2, sd = 1:4)))

# Overview over defined experiments:
tmp$problems
tmp$algorithms
summarizeExperiments(reg = tmp)
summarizeExperiments(reg = tmp, by = c("problem", "algorithm", "n"))
ids = findExperiments(prob.pars = (n == 50), reg = tmp)
print(unwrap(getJobPars(ids, reg = tmp)))

# Submit jobs
submitJobs(reg = tmp)
waitForJobs(reg = tmp)

# Reduce the results of algorithm a1
ids.mean = findExperiments(algo.name = "a1", reg = tmp)
reduceResults(ids.mean, fun = function(aggr, res, ...) c(aggr, res), reg = tmp)

# Join info table with all results and calculate mean of results
# grouped by n and algorithm
ids = findDone(reg = tmp)
pars = unwrap(getJobPars(ids, reg = tmp))
results = unwrap(reduceResultsDataTable(ids, fun = function(res) list(res = res), reg = tmp))
tab = ljoin(pars, results)
tab[, list(mres = mean(res)), by = c("n", "algorithm")]
```

---

makeJob *Jobs and Experiments*

Description

Jobs and Experiments are abstract objects which hold all information necessary to execute a single computational job for a Registry or ExperimentRegistry, respectively.

They can be created using the constructor makeJob which takes a single job id. Jobs and Experiments are passed to reduce functions like reduceResults. Furthermore, Experiments can be used in the functions of the Problem and Algorithm. Jobs and Experiments hold these information:

`job.id` Job ID as integer.

`pars` Job parameters as named list. For `ExperimentRegistry`, the parameters are divided into the sublists "prob.pars" and "algo.pars".

`seed` Seed which is set via `doJobCollection` as scalar integer.

`resources` Computational resources which were set for this job as named list.

`external.dir` Path to a directory which is created exclusively for this job. You can store external files here. Directory is persistent between multiple restarts of the job and can be cleaned by calling `resetJobs`.

`fun` Job only: User function passed to `batchMap`.

`prob.name` Experiments only: Problem id.

`algo.name` Experiments only: Algorithm id.

`problem` Experiments only: `Problem`.

`instance` Experiments only: Problem instance.

`algorithm` Experiments only: `Algorithm`.

`repl` Experiments only: Replication number.

Note that the slots "pars", "fun", "algorithm" and "problem" lazy-load required files from the file system and construct the object on the first access. The realizations are cached for all slots except "instance" (which might be stochastic).

Jobs and Experiments can be executed manually with `execJob`.

Usage

```
makeJob(id, reader = NULL, reg = getDefaultRegistry())
```

Arguments

`id`	[`integer(1)` or `data.table`] Single integer to specify the job or a `data.table` with column `job.id` and exactly one row.
`reader`	[`RDSReader` | `NULL`] Reader object to retrieve files. Used internally to cache reading from the file system. The default (`NULL`) does not make use of caching.
`reg`	[`Registry`] Registry. If not explicitly passed, uses the default registry (see `setDefaultRegistry`).

Value

`Job` | `Experiment` .

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(function(x, y) x + y, x = 1:2, more.args = list(y = 99), reg = tmp)
submitJobs(resources = list(foo = "bar"), reg = tmp)
job = makeJob(1, reg = tmp)
print(job)

# Get the parameters:
job$pars

# Get the job resources:
job$resources

# Execute the job locally:
execJob(job)
```

`makeJobCollection` *JobCollection Constructor*

Description

`makeJobCollection` takes multiple job ids and creates an object of class "JobCollection" which holds all necessary information for the calculation with `doJobCollection`. It is implemented as an environment with the following variables:

file.dir `file.dir` of the Registry.

work.dir: `work.dir` of the Registry.

job.hash Unique identifier of the job. Used to create names on the file system.

jobs `data.table` holding individual job information. See examples.

log.file Location of the designated log file for this job.

resources: Named list of of specified computational resources.

uri Location of the job description file (saved with `link[base]{saveRDS}` on the file system.

seed `integer(1)` Seed of the Registry.

packages `character` with required packages to load via `require`.

namespaces codecharacter with required packages to load via `requireNamespace`.

source `character` with list of files to source before execution.

load `character` with list of files to load before execution.

array.var `character(1)` of the array environment variable specified by the cluster functions.

array.jobs `logical(1)` signaling if jobs were submitted using `chunks.as.arrayjobs`.

If your ClusterFunctions uses a template, `brew` will be executed in the environment of such a collection. Thus all variables available inside the job can be used in the template.

Usage

```
makeJobCollection(ids = NULL, resources = list(),
  reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
resources	[list] Named list of resources. Default is list().
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

JobCollection .

See Also

Other JobCollection: doJobCollection

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE, packages = "methods")
batchMap(identity, 1:5, reg = tmp)

# resources are usually set in submitJobs()
jc = makeJobCollection(1:3, resources = list(foo = "bar"), reg = tmp)
ls(jc)
jc$resources
```

---

makeRegistry *Registry Constructor*

Description

makeRegistry constructs the inter-communication object for all functions in batchtools. All communication transactions are processed via the file system: All information required to run a job is stored as JobCollection in a file in the a subdirectory of the file.dir directory. Each jobs stores its results as well as computational status information (start time, end time, error message, ...) also on the file system which is regular merged parsed by the master using syncRegistry. After integrating the new information into the Registry, the Registry is serialized to the file system via saveRegistry. Both syncRegistry and saveRegistry are called whenever required internally.

Therefore it should be safe to quit the R session at any time. Work can later be resumed by calling loadRegistry which de-serializes the registry from the file system.

The registry created last is saved in the package namespace (unless make.default is set to FALSE) and can be retrieved via getDefaultRegistry.

Canceled jobs and jobs submitted multiple times may leave stray files behind. These can be swept using sweepRegistry. clearRegistry completely erases all jobs from a registry, including log files and results, and thus allows you to start over.

Usage

```
makeRegistry(file.dir = "registry", work.dir = getwd(),
  conf.file = findConfFile(), packages = character(0L),
  namespaces = character(0L), source = character(0L),
  load = character(0L), seed = NULL, make.default = TRUE)
```

Arguments

file.dir	[character(1)] Path where all files of the registry are saved. Default is directory "registry" in the current working directory. The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well. If you pass NA, a temporary directory will be used. This way, you can create disposable registries for btlapply or examples. By default, the temporary directory tempdir() will be used. If you want to use another directory, e.g. a directory which is shared between nodes, you can set it in your configuration file by setting the variable temp.dir.
work.dir	[character(1)] Working directory for R process for running jobs. Defaults to the working directory currently set during Registry construction (see getwd). loadRegistry uses the stored work.dir, but you may also explicitly overwrite it, e.g., after switching to another system. The provided path will get normalized unless it is given relative to the home directory (i.e., starting with "~"). Note that some templates do not handle relative paths well.
conf.file	[character(1)] Path to a configuration file which is sourced while the registry is created. In the configuration file you can define how **batchtools** interacts with the system via ClusterFunctions. Separating the configuration of the underlying host system from the R code allows to easily move computation to another site. The file lookup is implemented in the internal (but exported) function findConfFile which returns the first file found of the following candidates: 1. File "batchtools.conf.R" in the path specified by the environment variable "R_BATCHTOOLS_SEARCH_PATH". 2. File "batchtools.conf.R" in the current working directory. 3. File "config.R" in the user configuration directory as reported by rappdirs::user_config_dir("ba (depending on OS, e.g., on linux this usually resolves to "~/.config/batchtools/config.R").

4. ".batchtools.conf.R" in the home directory ("~").
5. "config.R" in the site config directory as reported by `rappdirs::site_config_dir("batchtools")` (depending on OS). This file can be used for admins to set sane defaults for a computation site.

Set to `NA` if you want to suppress reading any configuration file. If a configuration file is found, it gets sourced inside the environment of the registry after the defaults for all variables are set. Therefore you can set and overwrite slots, e.g. `default.resources = list(walltime = 3600)` to set default resources or "max.concurrent.jobs" to limit the number of jobs allowed to run simultaneously on the system.

`packages` [character]
Packages that will always be loaded on each node. Uses `require` internally. Default is `character(0)`.

`namespaces` [character]
Same as `packages`, but the packages will not be attached. Uses `requireNamespace` internally. Default is `character(0)`.

`source` [character]
Files which should be sourced on the slaves prior to executing a job. Calls `sys.source` using the `.GlobalEnv`.

`load` [character]
Files which should be loaded on the slaves prior to executing a job. Calls `load` using the `.GlobalEnv`.

`seed` [integer(1)]
Start seed for jobs. Each job uses the (`seed` + `job.id`) as seed. Default is a random number in the range [1, `.Machine$integer.max/2`].

`make.default` [logical(1)]
If set to `TRUE`, the created registry is saved inside the package namespace and acts as default registry. You might want to switch this off if you work with multiple registries simultaneously. Default is `TRUE`.

Details

Currently **batchtools** understands the following options set via the configuration file:

`cluster.functions`: As returned by a constructor, e.g. `makeClusterFunctionsSlurm`.

`default.resources`: List of resources to use. Will be overruled by resources specified via `submitJobs`.

`temp.dir`: Path to directory to use for temporary registries.

`sleep`: Custom sleep function. See `waitForJobs`.

`expire.after`: Number of iterations before treating jobs as expired in `waitForJobs`.

Value

`environment` of class "Registry" with the following slots:

`file.dir` **[path** :] File directory.

`work.dir` **[path** :] Working directory.

`temp.dir` **[path** :] Temporary directory. Used if `file.dir` is `NA` to create temporary registries.

`packages` **[character()** :] Packages to load on the slaves.

`namespaces` **[character()** :] Namespaces to load on the slaves.

`seed` **[integer(1)** :] Registry seed. Before each job is executed, the seed `seed + job.id` is set.

`cluster.functions` **[cluster.functions** :] Usually set in your `conf.file`. Set via a call to `makeClusterFunctions`. See example.

`default.resources` **[named list()** :] Usually set in your `conf.file`. Named list of default resources.

`max.concurrent.jobs` **[integer(1)** :] Usually set in your `conf.file`. Maximum number of concurrent jobs for a single user and current registry on the system. `submitJobs` will try to respect this setting. The resource "max.concurrent.jobs" has higher precedence.

`defs` **[data.table** :] Table with job definitions (i.e. parameters).

`status` **[data.table** :] Table holding information about the computational status. Also see `getJobStatus`.

`resources` **[data.table** :] Table holding information about the computational resources used for the job. Also see `getJobResources`.

`tags` **[data.table** :] Table holding information about tags. See Tags.

`hash` **[character(1)** :] Unique hash which changes each time the registry gets saved to the file system. Can be utilized to invalidate the cache of **knitr**.

See Also

Other Registry: `clearRegistry`, `getDefaultRegistry`, `loadRegistry`, `removeRegistry`, `saveRegistry`, `sweepRegistry`, `syncRegistry`

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
print(tmp)

# Set cluster functions to interactive mode and start jobs in external R sessions
tmp$cluster.functions = makeClusterFunctionsInteractive(external = TRUE)

# Change packages to load
tmp$packages = c("MASS")
saveRegistry(reg = tmp)
```

---

makeSubmitJobResult *Create a SubmitJobResult*

Description

This function is only intended for use in your own cluster functions implementation.

Use this function in your implementation of `makeClusterFunctions` to create a return value for the `submitJob` function.

Usage

```
makeSubmitJobResult(status, batch.id, log.file = NA_character_,
  msg = NA_character_)
```

Arguments

status	[integer(1)] Launch status of job. 0 means success, codes between 1 and 100 are temporary errors and any error greater than 100 is a permanent failure.
batch.id	[character()] Unique id of this job on batch system, as given by the batch system. Must be globally unique so that the job can be terminated using just this information. For array jobs, this may be a vector of length equal to the number of jobs in the array.
log.file	[character()] Log file. If NA, defaults to [job.hash].log. Some cluster functions set this for array jobs.
msg	[character(1)] Optional error message in case status is not equal to 0. Default is "OK", "TEMPERROR", "ERROR", depending on status.

Value

SubmitJobResult . A list, containing status, batch.id and msg.

See Also

Other ClusterFunctionsHelper: cfBrewTemplate, cfHandleUnknownSubmitError, cfKillJob, cfReadBrewTemplate, makeClusterFunctions, runOSCommand

---

reduceResults *Reduce Results*

Description

A version of Reduce for Registry objects which iterates over finished jobs and aggregates them. All jobs must have terminated, an error is raised otherwise.

Usage

```
reduceResults(fun, ids = NULL, init, ..., reg = getDefaultRegistry())
```

Arguments

fun	[function] A function to reduce the results. The result of previous iterations (or the init) will be passed as first argument, the result of of the i-th iteration as second. See Reduce for some examples. If the function has the formal argument "job", the Job/Experiment is also passed to the function (named).
ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of findDone. Invalid ids are ignored.
init	[ANY] Initial element, as used in Reduce. If missing, the reduction uses the result of the first job as init and the reduction starts with the second job.
...	[ANY] Additional arguments passed to function fun.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

Aggregated results in the same order as provided ids. Return type depends on the user function. If ids is empty, reduceResults returns init (if available) or NULL otherwise.

Note

If you have thousands of jobs, disabling the progress bar (options(batchtools.progress = FALSE)) can significantly increase the performance.

See Also

Other Results: batchMapResults, loadResult, reduceResultsList

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(function(a, b) list(sum = a+b, prod = a*b), a = 1:3, b = 1:3, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)

# Extract element sum from each result
reduceResults(function(aggr, res) c(aggr, res$sum), init = list(), reg = tmp)

# Aggregate element sum via '+'
reduceResults(function(aggr, res) aggr + res$sum, init = 0, reg = tmp)

# Aggregate element prod via '*' where parameter b < 3
reduce = function(aggr, res, job) {
  if (job$pars$b >= 3)
```

```
    return(aggr)
  aggr * res$prod
}
reduceResults(reduce, init = 1, reg = tmp)

# Reduce to data.frame() (inefficient, use reduceResultsDataTable() instead)
reduceResults(rbind, init = data.frame(), reg = tmp)

# Reduce to data.frame by collecting results first, then utilize vectorization of rbind:
res = reduceResultsList(fun = as.data.frame, reg = tmp)
do.call(rbind, res)

# Reduce with custom combine function:
comb = function(x, y) list(sum = x$sum + y$sum, prod = x$prod * y$prod)
reduceResults(comb, reg = tmp)

# The same with neutral element NULL
comb = function(x, y) if (is.null(x)) y else list(sum = x$sum + y$sum, prod = x$prod * y$prod)
reduceResults(comb, init = NULL, reg = tmp)

# Alternative: Reduce in list, reduce manually in a 2nd step
res = reduceResultsList(reg = tmp)
Reduce(comb, res)
```

`reduceResultsList` *Apply Functions on Results*

Description

Applies a function on the results of your finished jobs and thereby collects them in a `list` or `data.table`. The later requires the provided function to return a list (or `data.frame`) of scalar values. See `rbindlist` for features and limitations of the aggregation.

If not all jobs are terminated, the respective result will be `NULL`.

Usage

```
reduceResultsList(ids = NULL, fun = NULL, ..., missing.val,
  reg = getDefaultRegistry())

reduceResultsDataTable(ids = NULL, fun = NULL, ..., missing.val,
  reg = getDefaultRegistry())
```

Arguments

`ids`	[`data.frame` or `integer`] A `data.frame` (or `data.table`) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of `findDone`. Invalid ids are ignored.

fun	[function] Function to apply to each result. The result is passed unnamed as first argument. If NULL, the identity is used. If the function has the formal argument "job", the Job/Experiment is also passed to the function.
...	[ANY] Additional arguments passed to to function fun.
missing.val	[ANY] Value to impute as result for a job which is not finished. If not provided and a result is missing, an exception is raised.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

reduceResultsList returns a list of the results in the same order as the provided ids. reduceResultsDataTable returns a data.table with columns "job.id" and additional result columns created via rbindlist, sorted by "job.id".

Note

If you have thousands of jobs, disabling the progress bar (options(batchtools.progress = FALSE)) can significantly increase the performance.

See Also

reduceResults

Other Results: batchMapResults, loadResult, reduceResults

Examples

```
### Example 1 - reduceResultsList
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(function(x) x^2, x = 1:10, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)
reduceResultsList(fun = sqrt, reg = tmp)

### Example 2 - reduceResultsDataTable
tmp = makeExperimentRegistry(file.dir = NA, make.default = FALSE)

# add first problem
fun = function(job, data, n, mean, sd, ...) rnorm(n, mean = mean, sd = sd)
addProblem("rnorm", fun = fun, reg = tmp)

# add second problem
fun = function(job, data, n, lambda, ...) rexp(n, rate = lambda)
addProblem("rexp", fun = fun, reg = tmp)

# add first algorithm
```

```
fun = function(instance, method, ...) if (method == "mean") mean(instance) else median(instance)
addAlgorithm("average", fun = fun, reg = tmp)

# add second algorithm
fun = function(instance, ...) sd(instance)
addAlgorithm("deviation", fun = fun, reg = tmp)

# define problem and algorithm designs
prob.designs = algo.designs = list()
prob.designs$rnorm = CJ(n = 100, mean = -1:1, sd = 1:5)
prob.designs$rexp = data.table(n = 100, lambda = 1:5)
algo.designs$average = data.table(method = c("mean", "median"))
algo.designs$deviation = data.table()

# add experiments and submit
addExperiments(prob.designs, algo.designs, reg = tmp)
submitJobs(reg = tmp)

# collect results and join them with problem and algorithm paramters
res = ijoin(
  getJobPars(reg = tmp),
  reduceResultsDataTable(reg = tmp, fun = function(x) list(res = x))
)
unwrap(res, sep = ".")
```

`removeExperiments` *Remove Experiments*

Description

Remove Experiments from an `ExperimentRegistry`. This function automatically checks if any of the jobs to reset is either pending or running. However, if the implemented heuristic fails, this can lead to inconsistencies in the data base. Use with care while jobs are running.

Usage

```
removeExperiments(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[`data.frame` or `integer`] A `data.frame` (or `data.table`) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to no job. Invalid ids are ignored.
reg	[`ExperimentRegistry`] Registry. If not explicitly passed, uses the last created registry.

Value

`data.table` of removed job ids, invisibly.

See Also

Other Experiment: addExperiments, summarizeExperiments

---

removeRegistry *Remove a Registry from the File System*

Description

All files will be erased from the file system, including all results. If you wish to remove only intermediate files, use sweepRegistry.

Usage

```
removeRegistry(wait = 5, reg = getDefaultRegistry())
```

Arguments

wait	[numeric(1)] Seconds to wait before proceeding. This is a safety measure to not accidentally remove your precious files. Set to 0 in non-interactive scripts to disable this precaution.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

character(1) : Path of the deleted file directory.

See Also

Other Registry: clearRegistry, getDefaultRegistry, loadRegistry, makeRegistry, saveRegistry, sweepRegistry, syncRegistry

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
removeRegistry(0, tmp)
```

resetJobs *Reset the Computational State of Jobs*

Description

Resets the computational state of jobs in the Registry. This function automatically checks if any of the jobs to reset is either pending or running. However, if the implemented heuristic fails, this can lead to inconsistencies in the data base. Use with care while jobs are running.

Usage

```
resetJobs(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to no job. Invalid ids are ignored.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table of job ids which have been reset. See JoinTables for examples on working with job tables.

See Also

Other debug: getErrorMessages, getStatus, grepLogs, killJobs, showLog, testJob

runHook *Trigger Evaluation of Custom Function*

Description

Hooks allow to trigger functions calls on specific events. They can be specified via the ClusterFunctions and are triggered on the following events:

`pre.sync function(reg, fns, ...)`: Run before synchronizing the registry on the master. `fn` is the character vector of paths to the update files.

`post.sync function(reg, updates, ...)`: Run after synchronizing the registry on the master. `updates` is the data.table of processed updates.

`pre.submit.job function(reg, ...)`: Run before a job is successfully submitted to the scheduler on the master.

`post.submit.job function(reg, ...)`: Run after a job is successfully submitted to the scheduler on the master.

`pre.submit function(reg, ...)`: Run before any job is submitted to the scheduler.

`post.submit function(reg, ...)`: Run after a jobs are submitted to the schedule.

`pre.do.collection function(reg, cache, ...)`: Run before starting the job collection on the slave. `cache` is an internal cache object.

`post.do.collection function(reg, updates, cache, ...)`: Run after all jobs in the chunk are terminated on the slave. `updates` is a `data.table` of updates which will be merged with the `Registry` by the master. `cache` is an internal cache object.

`pre.kill function(reg, ids, ...)`: Run before any job is killed.

`post.kill function(reg, ids, ...)`: Run after jobs are killed. `ids` is the return value of `killJobs`.

Usage

```
runHook(obj, hook, ...)
```

Arguments

`obj`	[Registry | JobCollection] Registry which contains the ClusterFunctions with element "hooks" or a JobCollection which holds the subset of functions which are executed remotely.
`hook`	[`character(1)`] ID of the hook as string.
`...`	[ANY] Additional arguments passed to the function referenced by `hook`. See description.

Value

Return value of the called function, or `NULL` if there is no hook with the specified ID.

---

runOSCommand *Run OS Commands on Local or Remote Machines*

Description

This is a helper function to run arbitrary OS commands on local or remote machines. The interface is similar to `system2`, but it always returns the exit status *and* the output.

Usage

```
runOSCommand(sys.cmd, sys.args = character(0L), stdin = "",
  nodename = "localhost")
```

Arguments

sys.cmd	[character(1)] Command to run.
sys.args	[character] Arguments for sys.cmd.
stdin	[character(1)] Argument passed to system2.
nodename	[character(1)] Name of the SSH node to run the command on. If set to "localhost" (default), the command is not piped through SSH.

Value

named list with "sys.cmd", "sys.args", "exit.code" (integer), "output" (character).

See Also

Other ClusterFunctionsHelper: cfBrewTemplate, cfHandleUnknownSubmitError, cfKillJob, cfReadBrewTemplate, makeClusterFunctions, makeSubmitJobResult

Examples

```
## Not run:
runOSCommand("ls")
runOSCommand("ls", "-al")
runOSCommand("notfound")

## End(Not run)
```

---

saveRegistry *Store the Regisgty to the File System*

Description

Stores the registry on the file system in its "file.dir" (specified for construction in makeRegistry, can be accessed via reg$file.dir). This function is usually called internally whenever needed.

Usage

```
saveRegistry(reg = getDefaultRegistry())
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

logical(1) : TRUE if the registry was saved, FALSE otherwise (if the registry is read-only).

See Also

Other Registry: clearRegistry, getDefaultRegistry, loadRegistry, makeRegistry, removeRegistry, sweepRegistry, syncRegistry

---

showLog *Inspect Log Files*

Description

showLog opens the log in the pager. For customization, see file.show. getLog returns the log as character vector.

Usage

```
showLog(id, reg = getDefaultRegistry())

getLog(id, reg = getDefaultRegistry())
```

Arguments

id	[integer(1) or data.table] Single integer to specify the job or a data.table with column job.id and exactly one row.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

Nothing.

See Also

Other debug: getErrorMessages, getStatus, grepLogs, killJobs, resetJobs, testJob

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)

# Create some dummy jobs
fun = function(i) {
  if (i == 3) stop(i)
  if (i %% 2 == 1) warning("That's odd.")
}
```

```
ids = batchMap(fun, i = 1:5, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)
getStatus(reg = tmp)

writeLines(getLog(ids[1], reg = tmp))
## Not run:
showLog(ids[1], reg = tmp)

## End(Not run)

grepLogs(pattern = "warning", ignore.case = TRUE, reg = tmp)
```

`submitJobs` *Submit Jobs to the Batch Systems*

Description

Submits defined jobs to the batch system.

After submitting the jobs, you can use `waitForJobs` to wait for the termination of jobs or call `reduceResultsList`/`reduceResults` to collect partial results. The progress can be monitored with `getStatus`.

Usage

```
submitJobs(ids = NULL, resources = list(), sleep = NULL,
  reg = getDefaultRegistry())
```

Arguments

`ids`	[`data.frame` or `integer`] A `data.frame` (or `data.table`) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of `findNotSubmitted`. Invalid ids are ignored.
`resources`	[`named list`] Computational resources for the jobs to submit. The actual elements of this list (e.g. something like "walltime" or "nodes") depend on your template file, exceptions are outlined in the section 'Resources'. Default settings for a system can be set in the configuration file by defining the named list `default.resources`. Note that these settings are merged by name, e.g. merging `list(walltime = 300)` into `list(walltime = 400, memory = 512)` will result in `list(walltime = 300, memory = 512)`. Same holds for individual job resources passed as additional column of `ids` (c.f. section 'Resources').
`sleep`	[`function(i)` | `numeric(1)`] Parameter to control the duration to sleep between temporary errors. You can pass an absolute numeric value in seconds or a `function(i)` which returns the number of seconds to sleep in the `i`-th iteration between temporary errors. If not

provided (NULL), tries to read the value (number/function) from the configuration file (stored in reg$sleep) or defaults to a function with exponential backoff between 5 and 120 seconds.

reg [Registry]
Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with columns "job.id" and "chunk".

Resources

You can pass arbitrary resources to submitJobs() which then are available in the cluster function template. Some resources' names are standardized and it is good practice to stick to the following nomenclature to avoid confusion:

walltime: Upper time limit in seconds for jobs before they get killed by the scheduler. Can be passed as additional column as part of ids to set per-job resources.

memory: Memory limit in Mb. If jobs exceed this limit, they are usually killed by the scheduler. Can be passed as additional column as part of ids to set per-job resources.

ncpus: Number of (physical) CPUs to use on the slave. Can be passed as additional column as part of ids to set per-job resources.

omp.threads: Number of threads to use via OpenMP. Used to set environment variable "OMP_NUM_THREADS". Can be passed as additional column as part of ids to set per-job resources.

pp.size: Maximum size of the pointer protection stack, see Memory.

blas.threads: Number of threads to use for the BLAS backend. Used to set environment variables "MKL_NUM_THREADS" and "OPENBLAS_NUM_THREADS". Can be passed as additional column as part of ids to set per-job resources.

measure.memory: Enable memory measurement for jobs. Comes with a small runtime overhead.

chunks.as.arrayjobs: Execute chunks as array jobs.

pm.backend: Start a **parallelMap** backend on the slave.

foreach.backend: Start a **foreach** backend on the slave.

clusters: Resource used for Slurm to select the set of clusters to run sbatch/squeue/scancel on.

Chunking of Jobs

Multiple jobs can be grouped (chunked) together to be executed sequentially on the batch system as a single batch job. This is especially useful to avoid overburding the scheduler by submitting thousands of jobs simultaneously. To chunk jobs together, job ids must be provided as data.frame with columns "job.id" and "chunk" (integer). All jobs with the same chunk number will be executed sequentially inside the same batch job. The utility functions chunk, binpack and lpt can assist in grouping jobs.

Array Jobs

If your cluster supports array jobs, you can set the resource `chunks.as.arrayjobs` to `TRUE` in order to execute chunks as job arrays on the cluster. For each chunk of size `n`, **batchtools** creates a `JobCollection` of (possibly heterogeneous) jobs which is submitted to the scheduler as a single array job with `n` repetitions. For each repetition, the `JobCollection` is first read from the file system, then subsetted to the `i`-th job using the environment variable `reg$cluster.functions$array.var` (depending on the cluster backend, defined automatically) and finally executed.

Order of Submission

Jobs are submitted in the order of chunks, i.e. jobs which have chunk number `sort(unique(ids$chunk))[1]` first, then jobs with chunk number `sort(unique(ids$chunk))[2]` and so on. If no chunks are provided, jobs are submitted in the order of `ids$job.id`.

Limiting the Number of Jobs

If requested, `submitJobs` tries to limit the number of concurrent jobs of the user by waiting until jobs terminate before submitting new ones. This can be controlled by setting "max.concurrent.jobs" in the configuration file (see `Registry`) or by setting the resource "max.concurrent.jobs" to the maximum number of jobs to run simultaneously. If both are set, the setting via the resource takes precedence over the setting in the configuration.

Measuring Memory

Setting the resource `measure.memory` to `TRUE` turns on memory measurement: `gc` is called directly before and after the job and the difference is stored in the internal database. Note that this is just a rough estimate and does neither work reliably for external code like C/C++ nor in combination with threading.

Inner Parallelization

Inner parallelization is typically done via threading, sockets or MPI. Two backends are supported to assist in setting up inner parallelization.

The first package is **parallelMap**. If you set the resource "pm.backend" to "multicore", "socket" or "mpi", `parallelStart` is called on the slave before the first job in the chunk is started and `parallelStop` is called after the last job terminated. This way, the resources for inner parallelization can be set and get automatically stored just like other computational resources. The function provided by the user just has to call `parallelMap` to start parallelization using the preconfigured backend.

To control the number of CPUs, you have to set the resource `ncpus`. Otherwise `ncpus` defaults to the number of available CPUs (as reported by (see `detectCores`)) on the executing machine for multicore and socket mode and defaults to the return value of `mpi.universe.size-1` for MPI. Your template must be set up to handle the parallelization, e.g. request the right number of CPUs or start R with `mpirun`. You may pass further options like `level` to `parallelStart` via the named list "pm.opts".

The second supported parallelization backend is **foreach**. If you set the resource "foreach.backend" to "seq" (sequential mode), "parallel" (**doParallel**) or "mpi" (**doMPI**), the requested **foreach** back-

end is automatically registered on the slave. Again, the resource ncpus is used to determine the number of CPUs.

Neither the namespace of **parallelMap** nor the namespace **foreach** are attached. You have to do this manually via library or let the registry load the packages for you.

Note

If you a large number of jobs, disabling the progress bar (options(batchtools.progress = FALSE)) can significantly increase the performance of submitJobs.

Examples

```
### Example 1: Submit subsets of jobs
tmp = makeRegistry(file.dir = NA, make.default = FALSE)

# toy function which fails if x is even and an input file does not exists
fun = function(x, fn) if (x %% 2 == 0 && !file.exists(fn)) stop("file not found") else x

# define jobs via batchMap
fn = tempfile()
ids = batchMap(fun, 1:20, reg = tmp, fn = fn)

# submit some jobs
ids = 1:10
submitJobs(ids, reg = tmp)
waitForJobs(ids, reg = tmp)
getStatus(reg = tmp)

# create the required file and re-submit failed jobs
file.create(fn)
submitJobs(findErrors(ids, reg = tmp), reg = tmp)
getStatus(reg = tmp)

# submit remaining jobs which have not yet been submitted
ids = findNotSubmitted(reg = tmp)
submitJobs(ids, reg = tmp)
getStatus(reg = tmp)

# collect results
reduceResultsList(reg = tmp)

### Example 2: Using memory measurement
tmp = makeRegistry(file.dir = NA, make.default = FALSE)

# Toy function which creates a large matrix and returns the column sums
fun = function(n, p) colMeans(matrix(runif(n*p), n, p))

# Arguments to fun:
args = CJ(n = c(1e4, 1e5), p = c(10, 50)) # like expand.grid()
print(args)
```

```
# Map function to create jobs
ids = batchMap(fun, args = args, reg = tmp)

# Set resources: enable memory measurement
res = list(measure.memory = TRUE)

# Submit jobs using the currently configured cluster functions
submitJobs(ids, resources = res, reg = tmp)

# Retrive information about memory, combine with parameters
info = ijoin(getJobStatus(reg = tmp)[, .(job.id, mem.used)], getJobPars(reg = tmp))
print(unwrap(info))

# Combine job info with results -> each job is aggregated using mean()
unwrap(ijoin(info, reduceResultsDataTable(fun = function(res) list(res = mean(res)), reg = tmp)))

### Example 3: Multicore execution on the slave
tmp = makeRegistry(file.dir = NA, make.default = FALSE)

# Function which sleeps 10 seconds, i-times
f = function(i) {
  parallelMap::parallelMap(Sys.sleep, rep(10, i))
}

# Create one job with parameter i=4
ids = batchMap(f, i = 4, reg = tmp)

# Set resources: Use parallelMap in multicore mode with 4 CPUs
# batchtools internally loads the namespace of parallelMap and then
# calls parallelStart() before the job and parallelStop() right
# after the job last job in the chunk terminated.
res = list(pm.backend = "multicore", ncpus = 4)

## Not run:
# Submit both jobs and wait for them
submitJobs(resources = res, reg = tmp)
waitForJobs(reg = tmp)

# If successfull, the running time should be ~10s
getJobTable(reg = tmp)[, .(job.id, time.running)]

# There should also be a note in the log:
grepLogs(pattern = "parallelMap", reg = tmp)

## End(Not run)
```

---

summarizeExperiments *Quick Summary over Experiments*

Description

Returns a frequency table of defined experiments. See ExperimentRegistry for an example.

Usage

```
summarizeExperiments(ids = NULL, by = c("problem", "algorithm"),
  reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
by	[character] Split the resulting table by columns of getJobPars.
reg	[ExperimentRegistry] Registry. If not explicitly passed, uses the last created registry.

Value

data.table of frequencies.

See Also

Other Experiment: addExperiments, removeExperiments

---

sweepRegistry *Check Consistency and Remove Obsolete Information*

Description

Canceled jobs and jobs submitted multiple times may leave stray files behind. This function checks the registry for consistency and removes obsolete files and redundant data base entries.

Usage

```
sweepRegistry(reg = getDefaultRegistry())
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

See Also

Other Registry: clearRegistry, getDefaultRegistry, loadRegistry, makeRegistry, removeRegistry, saveRegistry, syncRegistry

syncRegistry *Synchronize the Registry*

Description

Parses update files written by the slaves to the file system and updates the internal data base.

Usage

```
syncRegistry(reg = getDefaultRegistry())
```

Arguments

reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

logical(1) : TRUE if the state has changed, FALSE otherwise.

See Also

Other Registry: clearRegistry, getDefaultRegistry, loadRegistry, makeRegistry, removeRegistry, saveRegistry, sweepRegistry

Tags *Add or Remove Job Tags*

Description

Add and remove arbitrary tags to jobs.

Usage

```
addJobTags(ids = NULL, tags, reg = getDefaultRegistry())

removeJobTags(ids = NULL, tags, reg = getDefaultRegistry())

getUsedJobTags(ids = NULL, reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all jobs. Invalid ids are ignored.
tags	[character] Tags to add or remove as strings. Each tag may consist of letters, numbers, underscore and dots (pattern "^[[:alnum:]_.]+").
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

data.table with job ids affected (invisible).

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
ids = batchMap(sqrt, x = -3:3, reg = tmp)

# Add new tag to all ids
addJobTags(ids, "needs.computation", reg = tmp)
getJobTags(reg = tmp)

# Add more tags
addJobTags(findJobs(x < 0, reg = tmp), "x.neg", reg = tmp)
addJobTags(findJobs(x > 0, reg = tmp), "x.pos", reg = tmp)
getJobTags(reg = tmp)

# Submit first 5 jobs and remove tag if successful
ids = submitJobs(1:5, reg = tmp)
if (waitForJobs(reg = tmp))
  removeJobTags(ids, "needs.computation", reg = tmp)
getJobTags(reg = tmp)

# Grep for warning message and add a tag
addJobTags(grepLogs(pattern = "NaNs produced", reg = tmp), "div.zero", reg = tmp)
getJobTags(reg = tmp)

# All tags where tag x.neg is set:
ids = findTagged("x.neg", reg = tmp)
getUsedJobTags(ids, reg = tmp)
```

---

testJob *Run Jobs Interactively*

Description

Starts a single job on the local machine.

Usage

```
testJob(id, external = FALSE, reg = getDefaultRegistry())
```

Arguments

id	[integer(1) or data.table] Single integer to specify the job or a data.table with column job.id and exactly one row.
external	[logical(1)] Run the job in an external R session? If TRUE, starts a fresh R session on the local machine to execute the with execJob. You will not be able to use debug tools like traceback or browser. If external is set to FALSE (default) on the other hand, testJob will execute the job in the current R session and the usual debugging tools work. However, spotting missing variable declarations (as they are possibly resolved in the global environment) is impossible. Same holds for missing package dependency declarations.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

Returns the result of the job if successful.

See Also

Other debug: getErrorMessages, getStatus, grepLogs, killJobs, resetJobs, showLog

Examples

```
tmp = makeRegistry(file.dir = NA, make.default = FALSE)
batchMap(function(x) if (x == 2) xxx else x, 1:2, reg = tmp)
testJob(1, reg = tmp)
## Not run:
testJob(2, reg = tmp)

## End(Not run)
```

unwrap *Unwrap Nested Data Frames*

Description

Some functions (e.g., getJobPars, getJobResources or reduceResultsDataTable return a data.table with columns of type list. These columns can be unnested/unwrapped with this function. The contents of these columns will be transformed to a data.table and cbind-ed to the input data.frame x, replacing the original nested column.

Usage

```
unwrap(x, cols = NULL, sep = NULL)

flatten(x, cols = NULL, sep = NULL)
```

Arguments

x	[data.frame | data.table] Data frame to flatten.
cols	[character] Columns to consider for this operation. If set to NULL (default), will operate on all columns of type "list".
sep	[character(1)] If NULL (default), the column names of the additional columns will re-use the names of the nested list/data.frame. This may lead to name clashes. If you provide sep, the variable column name will be constructed as "[column name of x][sep][inner name]".

Value

data.table .

Note

There is a name clash with function flatten in package **purrr**. The function flatten is discouraged to use for this reason in favor of unwrap.

Examples

```
x = data.table(
  id = 1:3,
  values = list(list(a = 1, b = 3), list(a = 2, b = 2), list(a = 3))
)
unwrap(x)
unwrap(x, sep = ".")
```

`waitForJobs` *Wait for Termination of Jobs*

Description

This function simply waits until all jobs are terminated.

Usage

```
waitForJobs(ids = NULL, sleep = NULL, timeout = 604800,
  expire.after = NULL, stop.on.error = FALSE, stop.on.expire = FALSE,
  reg = getDefaultRegistry())
```

Arguments

ids	[data.frame or integer] A data.frame (or data.table) with a column named "job.id". Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to the return value of findSubmitted. Invalid ids are ignored.
sleep	[function(i) | numeric(1)] Parameter to control the duration to sleep between queries. You can pass an absolute numeric value in seconds or a function(i) which returns the number of seconds to sleep in the i-th iteration. If not provided (NULL), tries to read the value (number/function) from the configuration file (stored in reg$sleep) or defaults to a function with exponential backoff between 5 and 120 seconds.
timeout	[numeric(1)] After waiting timeout seconds, show a message and return FALSE. This argument may be required on some systems where, e.g., expired jobs or jobs on hold are problematic to detect. If you don't want a timeout, set this to Inf. Default is 604800 (one week).
expire.after	[integer(1)] Jobs count as "expired" if they are not found on the system but have not communicated back their results (or error message). This frequently happens on managed system if the scheduler kills a job because the job has hit the walltime or request more memory than reserved. On the other hand, network file systems often require several seconds for new files to be found, which can lead to false positives in the detection heuristic. waitForJobs treats such jobs as expired after they have not been detected on the system for expire.after iterations. If not provided (NULL), tries to read the value from the configuration file (stored in reg$expire.after), and finally defaults to 3.
stop.on.error	[logical(1)] Immediately cancel if a job terminates with an error? Default is FALSE.
stop.on.expire	[logical(1)] Immediately cancel if jobs are detected to be expired? Default is FALSE. Expired jobs will then be ignored for the remainder of waitForJobs().

reg [Registry]
Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

logical(1) . Returns TRUE if all jobs terminated successfully and FALSE if either the timeout is reached or at least one job terminated with an exception or expired.

Worker *Create a Linux-Worker*

Description

R6Class to create local and remote linux workers.

Usage

```
Worker
```

Format

An R6Class generator object

Value

Worker .

Fields

nodename Host name. Set via constructor.

ncpus Number of CPUs. Set via constructor and defaults to a heuristic which tries to detect the number of CPUs of the machine.

max.load Maximum load average (of the last 5 min). Set via constructor and defaults to the number of CPUs of the machine.

status Status of the worker; one of "unknown", "available", "max.cpus" and "max.load".

Methods

new(nodename, ncpus, max.load) Constructor.

update(reg) Update the worker status.

list(reg) List running jobs.

start(reg, fn, outfile) Start job collection in file "fn" and output to "outfile".

kill(reg, batch.id) Kill job matching the "batch.id".

Examples

```
## Not run:
# create a worker for the local machine and use 4 CPUs.
Worker$new("localhost", ncpus = 4)

## End(Not run)
```

Index