Package 'batchtools'

April 22, 2017

Title Tools for Computation on Batch Systems

Version 0.9.3

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' (<http://www-03.ibm.com/systems/spectrum-computing/products/lsf/>), '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

Depends R (>= 3.0.0), data.table (>= 1.9.8)

Imports backports (>= 1.0.4), base64url (>= 1.1), brew, checkmate (>= 1.8.2), digest (>= 0.6.9), parallel, progress (>= 1.1.1), R6, rappdirs, stats, stringi, utils

Suggests debugme, e1071, knitr, parallelMap, ranger, rmarkdown, rpart, snow, testthat

VignetteBuilder knitr

RoxygenNote 6.0.1

Author Michel Lang [cre, aut],
Bernd Bischl [aut],
Dirk Surmann [ctb]

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2017-04-21 22:50:18 UTC

R topics documented:

batchtools-package . . . 3
addAlgorithm . . . 4
addExperiments . . . 5
addProblem . . . 6
batchExport . . . 8
batchMap . . . 9
batchMapResults . . . 10
batchReduce . . . 12
batchtools-deprecated . . . 13
btlapply . . . 13
cfBrewTemplate . . . 14
cfHandleUnknownSubmitError . . . 15
cfKillJob . . . 16
cfReadBrewTemplate . . . 17
chunk . . . 17
chunkIds . . . 19
clearRegistry . . . 20
doJobCollection . . . 21
estimateRuntimes . . . 22
execJob . . . 24
findJobs . . . 24
getDefaultRegistry . . . 27
getErrorMessages . . . 27
getJobTable . . . 28
getStatus . . . 30
grepLogs . . . 31
JoinTables . . . 32
killJobs . . . 33
loadRegistry . . . 34
loadResult . . . 35
makeClusterFunctions . . . 36
makeClusterFunctionsDocker . . . 37
makeClusterFunctionsInteractive . . . 39
makeClusterFunctionsLSF . . . 40
makeClusterFunctionsMulticore . . . 41
makeClusterFunctionsOpenLava . . . 42
makeClusterFunctionsSGE . . . 43
makeClusterFunctionsSlurm . . . 44
makeClusterFunctionsSocket . . . 46
makeClusterFunctionsSSH . . . 47
makeClusterFunctionsTORQUE . . . 48

makeExperimentRegistry . . . 49
makeJob . . . 51
makeJobCollection . . . 53
makeRegistry . . . 54
makeSubmitJobResult . . . 57
reduceResults . . . 58
reduceResultsList . . . 59
removeExperiments . . . 61
removeRegistry . . . 62
resetJobs . . . 62
runHook . . . 63
runOSCommand . . . 64
saveRegistry . . . 65
showLog . . . 66
submitJobs . . . 67
summarizeExperiments . . . 69
sweepRegistry . . . 70
syncRegistry . . . 71
Tags . . . 71
testJob . . . 72
waitForJobs . . . 73
Worker . . . 74

Index **76**

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

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

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>

Authors:

- Bernd Bischl <bernd_bischl@gmx.de>

Other contributors:

- Dirk Surmann <surmann@statistik.tu-dortmund.de> [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. getAlgorithmIds can be used to retrieve the IDs of already algorithms.

Usage

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

removeAlgorithms(name, reg = getDefaultRegistry())

getAlgorithmIds(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".

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)
getProblemIds(reg = tmp)

addAlgorithm("a1", fun = function(job, data, instance) instance, reg = tmp)
getAlgorithmIds(reg = tmp)

removeAlgorithms("a1", reg = tmp)
getAlgorithmIds(reg = tmp)
```

`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".

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$rnorm = expand.grid(n = 100, mean = -1:1, sd = 1:5)
prob.designs$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)
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. getProblemIds can be used to retrieve the IDs of already defined problems.

Usage

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

removeProblems(name, reg = getDefaultRegistry())

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

Value

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

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)
getProblemIds(reg = tmp)

addAlgorithm("a1", fun = function(job, data, instance) instance, reg = tmp)
getAlgorithmIds(reg = tmp)

removeAlgorithms("a1", reg = tmp)
getAlgorithmIds(reg = tmp)
```

---

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 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.

See Also

Other Results: loadResult, reduceResultsList, reduceResults

Examples

```
# Source registry: calculate squre 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: map some results of first registry to calculate the square root
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 = 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, 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)
```

---

batchtools-deprecated *Deprecated function in the batchtools package*

Description

The following functions have been deprecated:

chunkIds	deprecated in favor of chunk, lpt and binpack

---

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 batch jobs. The elements of this list (e.g. something like "walltime" or "nodes") depend on your template file. See notes for reserved special resource names. Defaults can be stored in the configuration file by providing the named list default.resources. Settings in resources overwrite those in default.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. If debug mode is enabled (via environment variable "DEBUGME" set to "batchtools"), the file is stored at the same place as the corresponding job file in the "jobs"-subdir of your files directory, otherwise as tempfile on the local system.

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 exception is generated.

Usage

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

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.

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 or single string (containing newlines) 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)

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.

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])


# 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)
```

---

chunkIds *Chunk Jobs for Sequential Execution*

Description

This function is deprecated in favor of the more flexible chunk, lpt and binpack.

Usage

```
chunkIds(ids = NULL, n.chunks = NULL, chunk.size = NULL,
  group.by = character(0L), 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.

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.
group.by	[character(0)] If ids is a data.frame with additional columns (in addition to the required column "job.id"), then the chunking is performed using subgroups defined by the columns set in group.by. See example.
reg	[Registry] Registry. If not explicitly passed, uses the default registry (see setDefaultRegistry).

Value

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

See Also

chunk binpack lpt

---

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 = 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).

For a summarizing overview over the status, see `getStatus`. Note that `findOnSystem` and `findExpired` are somewhat heuristic and may report misleading results, depending on the state of the system and the `ClusterFunctions` implementation.

Usage

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

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

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, flatten = NULL, prefix = FALSE,
  reg = getDefaultRegistry())

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

getJobResources(ids = NULL, flatten = NULL, prefix = FALSE,
  reg = getDefaultRegistry())

getJobPars(ids = NULL, flatten = NULL, prefix = FALSE,
  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.
flatten	[logical(1)] Transform the job parameters and/or resource specifications to data frame columns? Defaults to TRUE if all parameters (or resources) are scalar atomics. Otherwise each row of the column will hold a named list. New columns will be named using prefix.
prefix	[logical(1)] If set to TRUE, the prefix "par." is used to name column names of parameters for a Registry and prefixes "prob.par." and "algo.par." are used to name the columns of a ExperimentRegistry. Resources are prefixed with "res.".
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.

memory 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. Possibly expanded to separate columns (see prefix.

resources List of computational resources set for this job. Possibly expanded to separate columns (see prefix.

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, flatten = FALSE)

# Job parameters:
getJobPars(reg = tmp, flatten = FALSE)

# 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.

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

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`

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.

Usage

```
loadRegistry(file.dir = getwd(), work.dir = NULL,
  conf.file = findConfFile(), make.default = TRUE, update.paths = 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. For example, you can set cluster functions or default resources in it. The script is executed inside the environment of the registry after the defaults for all variables are set, thus you can set and overwrite slots, e.g. `default.resources = list(walltime = 3600)` to set default resources.

The file lookup defaults to a heuristic which first tries to read "batchtools.conf.R" in the current working directory. If not found, it looks for a configuration file "config.R" in the OS dependent user configuration directory as reported by via `rappdirs::user_config_dir("batchtools", expand = FALSE)` (e.g., on linux this usually resolves to "~/.config/batchtools/config.R"). If this file is also not found, the heuristic finally tries to read the file ".batchtools.conf.R" in the home directory ("~"). Set to `character(0)` if you want to disable this heuristic.

`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.

`update.paths` [logical(1)]
If set to TRUE, the `file.dir` and `work.dir` will be updated in the registry. Note that this is likely to break computation on the system! Only do this if no jobs are currently running. Default is FALSE. If the provided `file.dir` does not match the stored `file.dir`, `loadRegistry` will return a registry in read-only mode.

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 = FALSE,
  scheduler.latency = 0, fs.latency = NA_real_, 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
: [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.

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 NA which disables the expensive heuristic 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 NA which disables the expensive heuristic 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 = NA_real_)
```

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 NA which disables the expensive heuristic 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 (http://www-03.ibm.com/systems/spectrum-computing/products/lsf/).

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

Here, the default for `template` is "lsf". Alternatively, the template itself can be provided as a single string (including at least one newline "\n").

`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 `NA` which disables the expensive heuristic 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 = NA_real_)
```

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 `NA` which disables the expensive heuristic 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 (http://www.openlava.org/).

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

Here, the default for `template` is "openlava". Alternatively, the template itself can be provided as a single string (including at least one newline "\n").

`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 NA which disables the expensive heuristic 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", 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 current working directory.
2. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
3. ".batchtools.[template].tmpl" in the home directory.
4. "[template].tmpl" in the package installation directory in the subfolder "templates".

Here, the default for `template` is "sge". Alternatively, the template itself can be provided as a single string (including at least one newline "\n").

`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 `NA` which disables the expensive heuristic 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", clusters = NULL,
  array.jobs = TRUE, 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 current working directory.
2. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
3. ".batchtools.[template].tmpl" in the home directory.
4. "[template].tmpl" in the package installation directory in the subfolder "templates".

Here, the default for `template` is "slurm". Alternatively, the template itself can be provided as a single string (including at least one newline "\n").

`clusters` [character(1)]
If multiple clusters are managed by one Slurm system, the name of one cluster has to be specified. If only one cluster is present, this argument may be omitted. Note that you should not select the cluster in your template file via `#SBATCH --clusters`.

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

`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 NA which disables the expensive heuristic 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 NA which disables the expensive heuristic 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 `NA` which disables the expensive heuristic 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 current working directory.
2. "[template].tmpl" in the user config directory (see `user_config_dir`); on linux this is usually "~/.config/batchtools/[template].tmpl".
3. ".batchtools.[template].tmpl" in the home directory.
4. "[template].tmpl" in the package installation directory in the subfolder "templates".

Here, the default for `template` is "torque". Alternatively, the template itself can be provided as a single string (including at least one newline "\n").

`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 `NA` which disables the expensive heuristic 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. For example, you can set cluster functions or default resources in it. The script is executed inside the environment of the registry after the defaults for all variables are set, thus you can set and overwrite slots, e.g. `default.resources = list(walltime = 3600)` to set default resources.
The file lookup defaults to a heuristic which first tries to read "batchtools.conf.R" in the current working directory. If not found, it looks for a configuration file "config.R" in the OS dependent user configuration directory as reported by via `rappdirs::user_config_dir("batchtools", expand = FALSE)` (e.g., on linux this usually resolves to "~/.config/batchtools/config.R"). If this file is also not found, the heuristic finally tries to read the file ".batchtools.conf.R" in the home directory ("~"). Set to `character(0)` if you want to disable this heuristic.

`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:
getProblemIds(reg = tmp)
getAlgorithmIds(reg = tmp)
summarizeExperiments(reg = tmp)
summarizeExperiments(reg = tmp, by = c("problem", "algorithm", "n"))
ids = findExperiments(prob.pars = (n == 50), reg = tmp)
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 = getJobPars(ids, reg = tmp)
results = 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, cache = 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.
cache	[Cache | NULL] Cache to retrieve files. Used internally.
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. For example, you can set cluster functions or default resources in it. The script is executed inside the environment of the registry after the defaults for all variables are set, thus you can set and overwrite slots, e.g. default.resources = list(walltime = 3600) to set default resources.

The file lookup defaults to a heuristic which first tries to read "batchtools.conf.R" in the current working directory. If not found, it looks for a configuration file "config.R" in the OS dependent user configuration directory as reported by via rappdirs::user_config_dir("batchtools", expand = FALSE) (e.g., on linux this usually resolves to "~/.config/batchtools/config.R"). If this file is also not found, the heuristic finally tries to read the file ".batchtools.conf.R" in the home directory ("~"). Set to character(0) if you want to disable this heuristic.

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

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.

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.

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.

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.
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. Default is the first result.
...	[ANY] Additional arguments passed to 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(x) x^2, x = 1:10, reg = tmp)
submitJobs(reg = tmp)
waitForJobs(reg = tmp)
reduceResults(function(x, y) c(x, y), reg = tmp)
reduceResults(function(x, y) c(x, sqrt(y)), init = numeric(0), reg = tmp)
```

---

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, ..., flatten = 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).
flatten	[logical(1)] Transform results into separate data.table columns. Defaults to TRUE if all results are (a list of) scalar atomics, Otherwise each row of the column will hold a named list.

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$rnorm = expand.grid(n = 100, mean = -1:1, sd = 1:5)
prob.designs$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
ijoin(
  getJobPars(reg = tmp),
  reduceResultsDataTable(reg = tmp, fun = function(x) list(res = x))
)
```

`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.

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

logical(1) : Success of unlink.

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 "exit.code" (integer) and "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.

If an additional column "chunk" is found in the table `ids`, jobs will be grouped accordingly to be executed sequentially on the same slave. The utility functions `chunk`, `binpack` and `lpt` can assist in grouping jobs. Jobs are submitted in the order of chunks, i.e. jobs which have chunk number `unique(ids$chunk)[1]` first, then jobs with chunk number `unique(ids$chunk)[2]` and so on. If no chunks are provided, jobs are submitted in the order of `ids$job.id`.

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 = default.sleep,
  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 batch jobs. The elements of this list (e.g. something like "walltime" or "nodes") depend on your template file. See notes for reserved special resource names. Defaults can be stored in the configuration file by providing the named list `default.resources`. Settings in `resources` overwrite those in `default.resources`.
`sleep`	[`function(i) | numeric(1)`] Function which returns the duration to sleep in the `i`-th iteration between temporary errors. Alternatively, you can pass a single positive numeric value.
`reg`	[`Registry`] Registry. If not explicitly passed, uses the default registry (see `setDefaultRegistry`).

Value

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

Note

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.

If your cluster supports array jobs, you can set the resource `chunks.as.arrayjobs` to `TRUE` in order to execute chunks as job arrays. To do so, the job must be repeated `nrow(jobs)` times via the cluster functions template. The function `doJobCollection` (which is called on the slave) now retrieves the repetition number from the environment and restricts the computation to the respective job in the `JobCollection`.

Furthermore, the package provides support for inner parallelization using threading, sockets or MPI via the package **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 used resources for inner parallelization are set in the same place as the resources for the outer parallelization done by **batchtools** and all resources get stored together in the `Registry`. The user function just has to call `parallelMap` to start parallelization using the preconfigured backend.

Note that you should set the resource `ncpus` to control the number of CPUs to use in **parallelMap**. `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. start R with `mpirun` or request the right number of CPUs. You may pass further options like `level` to `parallelStart` via the named list "pm.opts".

Also note that if you have thousands of jobs, disabling the progress bar (`options(batchtools.progress = FALSE)`) can significantly increase the performance of `submitJobs`.

Examples

```
### Example 1: 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 = expand.grid(n = c(1e4, 1e5), p = c(10, 50))
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, memory)], getJobPars(reg = tmp))
```

```
print(info)

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

### Example 2: 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. May use 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)
```

---

waitForJobs *Wait for Termination of Jobs*

Description

This function simply waits until all jobs are terminated.

Usage

```
waitForJobs(ids = NULL, sleep = default.sleep, timeout = 604800,
  stop.on.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 findSubmitted. Invalid ids are ignored.
sleep	[function(i) | numeric(1)] Function which returns the duration to sleep in the i-th iteration. Alternatively, you can pass a single positive numeric value.
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).
stop.on.error	[logical(1)] Immediately cancel if a job terminates with an error? Default is FALSE.
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.

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

∗Topic **datasets**
    Worker, 74
.GlobalEnv, *50*, *55*

addAlgorithm, 4, *7*
addExperiments, 5, *49*, *61*, *70*
addJobTags *(*Tags*)*, 71
addProblem, 6
ajoin *(*JoinTables*)*, 32
Algorithm, *49*, *51*, *52*
Algorithm *(*addAlgorithm*)*, 4

batchExport, 8
batchMap, 9, *10–13*, *51*
batchMapResults, 10, *36*, *58*, *60*
batchReduce, *10*, 12
batchtools *(*batchtools-package*)*, 3
batchtools-deprecated, 13
batchtools-package, 3
binpack, *13*, *20*, *22*, *67*
binpack *(*chunk*)*, 17
brew, *17*, *53*
browser, *73*
btlapply, 13, *34*, *49*, *55*
btmapply, *9*
btmapply *(*btlapply*)*, 13

cbind, *5*
cfBrewTemplate, 14, *16*, *17*, *37*, *57*, *65*
cfHandleUnknownSubmitError, *15*, 15, *16*, *17*, *37*, *57*, *65*
cfKillJob, *15*, *16*, 16, *17*, *36*, *37*, *57*, *65*
cfReadBrewTemplate, *15*, *16*, 17, *37*, *57*, *65*
chunk, *12–14*, 17, *20*, *67*
chunkIds, 19
clearRegistry, 20, *27*, *35*, *54*, *56*, *62*, *65*, *70*, *71*
ClusterFunctions, *24*, *38*, *39*, *41*, *43*, *44*, *46*, *47*, *49*, *53*, *63*, *64*
ClusterFunctions *(*makeClusterFunctions*)*, 36

data.frame, *5*, *11*, *19*, *20*, *25*, *27*, *29–32*, *34*, *53*, *58*, *59*, *61*, *63*, *67*, *70*, *72*, *74*
data.table, *5*, *6*, *10–12*, *19*, *20*, *22*, *25–31*, *33*, *34*, *53*, *58–61*, *63*, *67*, *70*, *72*, *74*
detectCores, *41*, *46*, *68*
difftime, *29*
doJobCollection, 21, *51*, *53*, *54*, *68*

estimateRuntimes, *18*, 22
execJob, 24, *52*, *73*
Experiment, *4*, *7*, *24*, *58*, *59*
Experiment *(*makeJob*)*, 51
ExperimentRegistry, *4*, *6*, *7*, *29*, *51*, *61*, *69*, *70*
ExperimentRegistry *(*makeExperimentRegistry*)*, 49

file.show, *66*
findDone, *11*, *58*, *59*
findDone *(*findJobs*)*, 24
findErrors, *27*
findErrors *(*findJobs*)*, 24
findExperiments *(*findJobs*)*, 24
findExpired *(*findJobs*)*, 24
findJobs, 24
findNotDone *(*findJobs*)*, 24
findNotStarted *(*findJobs*)*, 24
findNotSubmitted, *67*
findNotSubmitted *(*findJobs*)*, 24
findOnSystem, *34*
findOnSystem *(*findJobs*)*, 24
findQueued *(*findJobs*)*, 24
findRunning *(*findJobs*)*, 24
findStarted, *31*
findStarted *(*findJobs*)*, 24
findSubmitted, *74*
findSubmitted *(*findJobs*)*, 24

findTagged (findJobs), 24

gc, *68*
getAlgorithmIds (addAlgorithm), 4
getDefaultRegistry, *20*, 27, *35*, *54*, *56*, *62*, *65*, *70*, *71*
getErrorMessages, 27, *30*, *31*, *34*, *63*, *66*, *73*
getJobPars, *70*
getJobPars (getJobTable), 28
getJobResources, *56*
getJobResources (getJobTable), 28
getJobStatus, *56*
getJobStatus (getJobTable), 28
getJobTable, 28
getJobTags (getJobTable), 28
getLog (showLog), 66
getProblemIds (addProblem), 6
getStatus, *24*, *28*, 30, *31*, *34*, *63*, *66*, *67*, *73*
getUsedJobTags (Tags), 71
getwd, *34*, *49*, *55*
grepLogs, *28*, *30*, 31, *34*, *63*, *66*, *73*

Hook, *38*
Hook (runHook), 63
Hooks, *37*
Hooks (runHook), 63

ijoin (JoinTables), 32

Job, *4*, *7*, *9*, *24*, *58*, *59*
Job (makeJob), 51
JobCollection, *15*, *21*, *36*, *37*, *40*, *42*, *43*, *45*, *48*, *54*, *64*, *68*
JobCollection (makeJobCollection), 53
JoinTables, 32, *63*

killJobs, *28*, *30*, *31*, 33, *63*, *66*, *73*

lapply, *13*
list, *59*
ljoin (JoinTables), 32
load, *50*, *55*
loadRegistry, *20*, *27*, 34, *54*, *56*, *62*, *65*, *70*, *71*
loadResult, *9*, *11*, *12*, 35, *58*, *60*
lpt, *13*, *20*, *22*, *67*
lpt (chunk), 17

makeClusterFunctions, *15–17*, 36, *39*, *41–44*, *46*, *47*, *49*, *56*, *57*, *65*
makeClusterFunctionsDocker, *37*, 37, *39*, *41–44*, *46*, *47*, *49*
makeClusterFunctionsInteractive, *37*, *39*, 39, *41–44*, *46*, *47*, *49*
makeClusterFunctionsLSF, *37*, *39*, 40, *42–44*, *46*, *47*, *49*
makeClusterFunctionsMulticore, *37*, *39*, *41*, 41, *43*, *44*, *46*, *47*, *49*
makeClusterFunctionsOpenLava, *37*, *39*, *41*, *42*, 42, *44*, *46*, *47*, *49*
makeClusterFunctionsSGE, *37*, *39*, *41–43*, 43, *46*, *47*, *49*
makeClusterFunctionsSlurm, *37*, *39*, *41–44*, 44, *46*, *47*, *49*
makeClusterFunctionsSocket, *37*, *39*, *41–44*, *46*, 46, *47*, *49*
makeClusterFunctionsSSH, *37*, *39*, *41–44*, *46*, 47, *49*
makeClusterFunctionsTORQUE, *37*, *39*, *41–44*, *46*, *47*, 48
makeExperimentRegistry, 49
makeJob, *24*, 51
makeJobCollection, *21*, 53
makeRegistry, *13*, *20*, *27*, *35*, 54, *62*, *65*, *70*, *71*
makeSubmitJobResult, *15–17*, *37*, 57, *65*
Map, *9*
mapply, *13*
mpi.universe.size, *68*

ojoin (JoinTables), 32

parallelMap, *68*
parallelStart, *68*
parallelStop, *68*
POSIXct, *29*
print.RuntimeEstimate (estimateRuntimes), 22
Problem, *4*, *49*, *51*, *52*
Problem (addProblem), 6

R6Class, *74*
ranger, *22*
rbindlist, *59*, *60*
Reduce, *12*, *58*
reduceResults, *9*, *11*, *12*, *36*, *51*, 58, *60*, *67*
reduceResultsDataTable (reduceResultsList), 59
reduceResultsList, *9*, *11*, *12*, *36*, *58*, 59, *67*

Registry, *8*, *10–12*, *14–16*, *20*, *22*, *25*, *27*, *29–31*, *33–38*, *49*, *51–53*, *58*, *59*, *62–68*, *70–74*
Registry *(*makeRegistry*)*, 54
removeAlgorithms *(*addAlgorithm*)*, 4
removeExperiments, *6*, 61, *70*
removeJobTags *(*Tags*)*, 71
removeProblems *(*addProblem*)*, 6
removeRegistry, *20*, *27*, *35*, *56*, 62, *65*, *70*, *71*
require, *50*, *53*, *55*
requireNamespace, *50*, *53*, *55*
resetJobs, *28*, *30*, *31*, *34*, *51*, 62, *66*, *73*
rjoin *(*JoinTables*)*, 32
runHook, 63
runOSCommand, *15–17*, *37*, *57*, 64

saveRegistry, *20*, *27*, *35*, *54*, *56*, *62*, 65, *70*, *71*
setDefaultRegistry, *8*, *10–12*, *14–16*, *20*, *22*, *25*, *27*, *29–31*, *34*, *36*, *52*, *53*, *58*, *59*, *62*, *63*, *65–67*, *70–74*
setDefaultRegistry *(*getDefaultRegistry*)*, 27
showLog, *28*, *30*, *31*, *34*, *63*, 66, *73*
simplify2array, *14*
sink, *21*
sjoin *(*JoinTables*)*, 32
SubmitJobResult, *15*, *16*, *36*, *57*
SubmitJobResult *(*makeSubmitJobResult*)*, *57*
submitJobs, *9*, *12*, *14*, *17*, *21*, *37–40*, *42–45*, *48*, *56*, 67
summarizeExperiments, *6*, *61*, 69
sweepRegistry, *20*, *27*, *35*, *54*, *56*, *62*, *65*, 70, *71*
syncRegistry, *20*, *27*, *35*, *38*, *54*, *56*, *62*, *65*, *70*, 71
sys.source, *50*, *55*
system2, *64*

Tags, *28*, *56*, 71
tempdir, *34*, *49*, *55*
tempfile, *14*
testJob, *28*, *30*, *31*, *34*, *63*, *66*, 72
traceback, *73*

ujoin *(*JoinTables*)*, 32
unlink, *62*
user_config_dir, *40*, *42*, *44*, *45*, *48*

vector, *14*

waitForJobs, *13*, *67*, 73
Worker, *47*, *74*, 74