Package 'parallelize.dynamic'

July 2, 2014

Type Package

Title Automate parallelization of function calls by means of dynamic code analysis

Version 0.9-1

Date 2013-01-24

Author Stefan Boehringer <r-packages@s-boehringer.org>

Maintainer Stefan Boehringer <r-packages@s-boehringer.org>

Description Passing a given function name or a call to the parallelize/parallelize_call functions analyses and executes the code, if possible in parallel. Parallel code execution can be performed locally or on remote batch queuing systems.

License LGPL

Depends methods, tools, parallel

Encoding UTF-8

NeedsCompilation no

Repository CRAN

Date/Publication 2013-05-22 20:25:33

R topics documented:

`parallelize.dynamic-package`

Automate parallelization of function calls by means of dynamic code analysis

---

Description

Passing a given function name or a call to the parallelize/parallelize_call functions analyses and executes the code, if possible in parallel. Parallel code execution can be performed locally or on remote batch queuing systems.

Details

Package:	parallelize.dynamic
Type:	Package
Version:	0.9
Date:	2012-12-12
License:	LGPL
Depends:	methods, tools, parallel

Use `parallelize_initialize` to set up a configuration for performing parallel computations. After that, you can use `parallelize` and `parallelize_call` to run a dynamic analysis on given functions or function calls and execute parallel jobs resulting from this analysis. For the remote backend OGSremote, the current implmentation is expected to break on machines running operating systems from the Windows family on account of dependencies on system calls. The local backend snow should work on Windows. Patches are welcome to solve any Windows issues.

Author(s)

Stefan Böhringer

Maintainer: Stefan Böhringer <r-packages@s-boehringer.org>

References

R Journal article "Dynamic parallelization of R functions", submitted

See Also

parallel

`Apply` *Expose an apply-loop to parallelization*

Description

Replacing and apply/lapply/sapply call with a Apply/Lapply/Sapply call makes it amenable to analysis by the parallelize function that can determine dynamic parallelism in running code.

Usage

```
Apply(X, MARGIN, FUN, ...)
Lapply(l, .f, ..., Lapply_config = Lapply_getConfig(),
       Lapply_local = Lapply_config$local, Lapply_chunk = 1)
Sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

Arguments

`X`	See documentation for `apply`.
`MARGIN`	See documentation for `apply`.
`FUN`	See documentation for `sapply`.
`simplify`	See documentation for `sapply`.
`USE.NAMES`	See documentation for `sapply`.
`.f`	See documentation for `lapply`.
`l`	See documentation for `lapply`.
`Lapply_config`	See documentation for `parallelize_intialize`. Normally, this argument should be ignored.
`Lapply_local`	Force local execution. Normally, this argument should be ignored.
`Lapply_chunk`	Normally, this argument should be ignored.
`...`	See documentation for `apply`.

Details

Please refer to the documentation of apply/lapply/sapply for further documenation. The semantics of Apply/Lapply/Sapply are identical to apply/lapply/sapply. Using these functions implies that you want the parallelization mechanism to be applied to these loops.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

parallelize

Examples

```
r0 = sapply(1:10, function(x)x^2);
r1 = Sapply(1:10, function(x)x^2);
print(all(r0 == r1));
```

LapplyExecutionState-class

Class "LapplyExecutionState"

Description

An instance of this class reflects the entire lifetime of a dynamic parallelization.

Extends

All reference classes extend and inherit methods from "envRefClass".

Fields

sequenceNos: For each ramp-up, information of the Apply sequence number where parallelization starts and stops is stored together with nesting depth. This allows the Apply functions to switch to proper states. Each object on this stack represents a range of Apply calls.

rampUp: Ramp-up curser.

freezerClass: Name of the freezer class to be used.

freezers: Freezer objects for the different ramp-ups.

Methods

setFreezers(value): Accessor method for freezers slot.

getFreezers(): Accessor method for freezers slot.

setFreezerClass(value): Accessor method for freezerClass slot.

getFreezerClass(): Accessor method for freezerClass slot.

setRampUp(value): Accessor method for rampUp slot.

getRampUp(): Accessor method for rampUp slot.

setSequenceNos(value): Accessor method for sequenceNos slot.

getSequenceNos(): Accessor method for sequenceNos slot.

`currentFreezer()`: Topmost freezer object on the freezer stack.

`adjustCursor(state)`: Assign new value to topmost list from sequenceNos (during probing).

`isLastRampUp()`: Is the ramp-up curser located in the last ramp-up seen so far?

`skipToRampDown(state)`: In order to reach the last ramp-up (ramp-down) should we skip the given `state`?

`checkAgainstState(state)`: Is the given `state` located in the top-most range of the `sequenceNos` stack.

`rampUpForeFront()`: How many ramp-ups have been seen so far?

`resetCursor()`: Reset variable `rampUp` to one. This prepares for a re-execution of the function.

`incCursor()`: Indicate the switch to the next ramp-up.

`currentSentinel()`: Return last object from the `sequencNos` stack.

`pushSequenceForRampUp(sequenceNo, depth)`: Indicate the visiting of `Apply` call at sequence number `sequenceNo` and depth `depth`. The method automatically updates the `sequenceNos` stack to record `Apply` calls with maximal depth and accumulates calls at equal depth into a start/stop range.

`addSentinel()`: Create a new element on the `sequenceNos` stack.

`initialize(freezerClass, ...)`: Initialize the object.

Author(s)

Stefan Böhringer

Maintainer: Stefan Böhringer <r-packages@s-boehringer.org>

See Also

`LapplyFreezer-class`, `LapplyState-class`

Examples

```
showClass("LapplyExecutionState")
```

LapplyFreezer-class *Class* "LapplyFreezer"

Description

This class encapsulates storage of calls and their results. Interaction with this is done from backends and subclassing is only required if a new storage mechanism of unevaluated calls or results thereof is needed. The end user does not interact with this class.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Fields

`slots`: Each slot represents an Apply call, where a function is applied on a list. The slot records the arity of the list and the sequence number of the call.

`calls`: All calls to be parallelized.

`results`: Results of the calls.

Methods

`setResults(value)`: accessor method of results slot

`getResults()`: accessor method of results slot

`setCalls(value)`: accessor method of calls slot

`getCalls()`: accessor method of calls slot

`setSlots(value)`: accessor method of slots slot

`getSlots()`: accessor method of slots slot

`resultsForSequence(s)`: return results for `Apply` sequence number s

`finalizeResults()`: must be called exactly once before resultsForSequence is called. After all parllel computations are finished and their results are collected, this method should be called.

`unlistResults()`: Return flat list of results.

`pushResults(r)`: Push a new result. Order of push must be in the order of call generation in push.

`call(i)`: Return i-th call encapsulated as a list.

`N()`: Return number of calls generated so far.

`push(sequence, f, l, ..., envir__)`: Add calls from Apply-loop sequence to the freezer. `length(l)` calls will be generated.

`clear()`: Clear the freezer.

`initialize(...)`: Initialize the freezer.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

`LapplyPersistentFreezer-class`

Examples

```
showClass("LapplyFreezer")
```

LapplyPersistentFreezer-class

Class "LapplyPersistentFreezer"

Description

Subclass of LapplyFreezer that stores results on disk. See LapplyFreezer for more documentation.

Extends

Class "LapplyFreezer", directly.

All reference classes extend and inherit methods from "envRefClass".

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

LapplyFreezer-class

Examples

```
showClass("LapplyPersistentFreezer")
```

LapplyProbeState-class

Class "LapplyProbeState"

Description

This subclass of "LapplyState" tracks probing runs of the parallelization process.

Extends

Class "LapplyState", directly.

All reference classes extend and inherit methods from "envRefClass".

Note

See documentation of "LapplyState".

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

"LapplyState"

Examples

```
showClass("LapplyProbeState")
```

LapplyRunState-class *Class* "LapplyRunState"

Description

This subclass of "LapplyState" tracks running of code of the parallelization process.

Extends

Class "LapplyState", directly.

All reference classes extend and inherit methods from "envRefClass".

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

LapplyState

Examples

```
showClass("LapplyRunState")
```

LapplyState-class *Class* "LapplyState"

Description

This class is the base class for classes reflecting different stages of the parallelization process: probing and running.

Extends

All reference classes extend and inherit methods from "envRefClass".

Fields

`sequence`: Current sequence number of `Apply` calls.

`depth`: Current nesting level of `Apply` calls.

`runMode`: Currently running?

`probeMode`: Currently probing?

`max_depth`: Maxmimum of nesting to be considered.

Methods

`setMax_depth(value)`: Accessor method.

`getMax_depth()`: Accessor method.

`setProbeMode(value)`: Accessor method.

`getProbeMode()`: Accessor method.

`setRunMode(value)`: Accessor method.

`getRunMode()`: Accessor method.

`setDepth(value)`: Accessor method.

`getDepth()`: Accessor method.

`setSequence(value)`: Accessor method.

`getSequence()`: Accessor method.

`isEqualTo(s)`: Compare to other state for equalness.

`sequenceDec()`: Advance sequence number.

`sequenceInc()`: Un-advance sequence number.

`depthDec()`: Transition to deeper nesting level.

`depthInc()`: Transition to shallower nesting level.

`initialize(...)`: Initialize object.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

`LapplyExecutionState-class`, `LapplyRunState-class`

Examples

```
showClass("LapplyState")
```

Log *Log a message to stderr.*

Description

Log a message to stderr. Indicate a logging level to control verbosity.

Usage

```
Log(o, level = get("DefaultLogLevel", envir = Log_env__))
Log.setLevel(level = get("GlobalLogLevel", envir = Log_env__))
Log.level()
```

Arguments

o	Message to be printed.
level	If `Log.setLevel` was called with this value, subsequent calls to `Log` with values of `level` smaller or equal to this value will be printed.

Details

This function prints a message to stderr if the condition is met that a global log-level is set to greater or equal the value indicated by `level`. `Log.level` returns the current logging level.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

`Log.setLevel`, `~~~`

Examples

```
Log.setLevel(4);
Log('hello world', 4);
Log.setLevel(3);
Log('hello world', 4);
```

`parallelize` *Subject a function to dynamic parallelization*

Description

This function executes all necessary steps to perform a dynamic analysis of parallelism of a given function, create objects that encapsulate code that can be executed in parallel, transfer this to a execution backend which potentially is a remote target, recollect results and resume execution in a transparent way.

Usage

```
parallelize(.f, ..., Lapply_local = rget("Lapply_local", default = FALSE),
  parallelize_wait = TRUE)
parallelize_call(.call, ..., parallelize_wait = TRUE)
```

Arguments

`.f`	Function to be parallelized given as a function object.
`...`	Arguments passed to .f.
`Lapply_local`	Force local execution.
`parallelize_wait`	Force to poll completion of computation if backend returns asynchroneously
`.call`	Unevaluated call to be parallelized

Details

Function parallelize and parallelize_call both perform a dynamic parallelization of the computation of .f, i.e. parallelism is determined at run-time. Points of potential parallelism have to be indicated by the use of Apply/Sapply/Lapply (collectively Apply functions) instead of apply/sapply/lapply in existing code. The semantics of the new function is exactly the same as that of the original functions such that a simple upper-casing of these function calls makes existing programs amenable to parallelization. Parallelize will execute the function .f, recording the number of elements that are passed to Apply functions. Once a given threshold (degree of parallelization) is reached computation is stopped and remaining executions are done in parallel. A call parallelize_initialize function determines the precise mechanism of parallel execution and can be used to flexibly switch between different resources.

Value

The value returned is the result of the computation of function .f

Important details

- The package creates files in a given folder. This folder is not temporary as it might be needed across R sessions. md5-fingerprints of textual representations of function calls are used to create subfolders therein per `parallelize` call. Conflicts can be avoided by choosing different function names per `parallelize` call for parallelizing the same function several times. For example: `f0 = f1 = function(){42}; parallelize(f0); parallelize(f1);`
- In view of efficiency the parallize.dynamic package does not copy the whole workspace to all parallel jobs. Instead, a hopefully minimal environment is set up for each parallel job. This includes all parameters passed to the function in question together with variables defined in the closure. This leaves all functions undefined and it is therefore expected that all necessary function defintions are given in separate R-files or libraries which have to be specified in the `config` variable. These are then sourced/loaded into the parallel job. A way to dynamically create source files from function definitions is given in the Examples section.

Author(s)

Stefan Böhringer, <r-packages@s-boehringer.org>

See Also

Apply, Sapply, Lapply, parallelize_initialize

Examples

```
# code to be parallelized
parallel8 = function(e) log(1:e) %*% log(1:e);
parallel2 = function(e) rep(e, e) %*% 1:e * 1:e;
parallel1 = function(e) Lapply(rep(e, 15), parallel2);
parallel0 = function() {
  r = sapply(Lapply(1:50, parallel1),
    function(e)sum(as.vector(unlist(e))));
  r0 = Lapply(1:49, parallel8);
  r
}

# create file that can be sourced containing function definitions
# best practice is to define all needed functions in files that
# can be sourced. The function tempcodefile allows to create a
# temporary file with the defintion of given functions
codeFile = tempcodefile(c(parallel0, parallel1, parallel2, parallel8));

# definitions of clusters
Parallelize_config = list(max_depth = 5, parallel_count = 24, offline = FALSE,
backends = list(
  snow = list(localNodes = 2, splitN = 1, sourceFiles = codeFile),
  local = list(
    path = sprintf('%s/tmp/parallelize', tempdir())
  )
));
```

```
# initialize
parallelize_initialize(Parallelize_config, backend = 'local');

# perform parallelization
r0 = parallelize(parallel0);
print(r0);

# same
r1 = parallelize_call(parallel0());
print(r1);

# compare with native execution
parallelize_initialize(backend = 'off');
r2 = parallelize(parallel0);
print(r2);

# put on SNOW cluster
parallelize_initialize(Parallelize_config, backend = 'snow');
r3 = parallelize(parallel0);
print(r3);

# analyse parallelization
parallelize_initialize(Parallelize_config, backend = 'local');
Log.setLevel(5);
r4 = parallelize(parallel0);
Log.setLevel(6);
r5 = parallelize(parallel0);

print(sprintf('All results are the same is %s', as.character(
  all(sapply(list(r0, r1, r2, r3, r4, r5), function(l)all(l == r0)))
)));
```

`ParallelizeBackend-class`
Class "`ParallelizeBackend`"

Description

Base class for parallelization backends. Please refer to documentation of the methods individually for more complete documentation.

Objects from the Class

Objects can be created by calls of the form `new("ParallelizeBackend", config, signature)`.

Config is a list containing parameters and signature is a character string that uniquely identifies the computation that is to be parallelized.

Slots

`config`: Object of class `"list"` ~~

`offline`: Object of class `"logical"` ~~

`signature`: Object of class `"character"` ~~

Methods

finalizeParallelization `signature(self = "ParallelizeBackend")`: ...

getResult `signature(self = "ParallelizeBackend")`: ...

initialize `signature(.Object = "ParallelizeBackend")`: ...

initScheduling `signature(self = "ParallelizeBackend")`: ...

isSynchroneous `signature(self = "ParallelizeBackend")`: ...

lapply_dispatchFinalize `signature(self = "ParallelizeBackend")`: ...

lapply_dispatch `signature(self = "ParallelizeBackend")`: ...

lapply_results `signature(self = "ParallelizeBackend")`: ...

parallelize_backend `signature(self = "ParallelizeBackend")`: ...

performParallelizationStep `signature(self = "ParallelizeBackend")`: ...

pollParallelization `signature(self = "ParallelizeBackend")`: ...

restoreParallelizationState `signature(self = "ParallelizeBackend")`: ...

saveParallelizationState `signature(self = "ParallelizeBackend")`: ...

scheduleNextParallelization `signature(self = "ParallelizeBackend")`: ...

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

`ParallelizeBackendLocal`, `ParallelizeBackendSnow`, `ParallelizeBackendOGSremote`

Examples

```
showClass("ParallelizeBackend")
```

ParallelizeBackendLocal-class
Class "ParallelizeBackendLocal"

Description

Backend class implementing local execution

Objects from the Class

Objects can be created by calls of the form `new("ParallelizeBackendLocal", config, ...)`.

Slots

config: Object of class "list" ~~

offline: Object of class "logical" ~~

signature: Object of class "character" ~~

Extends

Class "ParallelizeBackend", directly.

Methods

initialize `signature(.Object = "ParallelizeBackendLocal")`: ...

lapply_dispatchFinalize `signature(self = "ParallelizeBackendLocal")`: ...

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

ParallelizeBackend, ParallelizeBackendSnow, ParallelizeBackendOGSremote

Examples

```
showClass("ParallelizeBackendLocal")
```

ParallelizeBackendOGS-class
Class "ParallelizeBackendOGS"

Description

Backend class implmenting Open Grid Scheduler support

Objects from the Class

Objects can be created by calls of the form .

Slots

jids: Object of class "ParallelizeBackendOGSstate" ~~

config: Object of class "list" ~~

offline: Object of class "logical" ~~

signature: Object of class "character" ~~

Extends

Class "ParallelizeBackend", directly.

Methods

finalizeParallelization signature(self = "ParallelizeBackendOGS"): ...

initialize signature(.Object = "ParallelizeBackendOGS"): ...

initScheduling signature(self = "ParallelizeBackendOGS"): ...

lapply_dispatchFinalize signature(self = "ParallelizeBackendOGS"): ...

pollParallelization signature(self = "ParallelizeBackendOGS"): ...

restoreParallelizationState signature(self = "ParallelizeBackendOGS"): ...

scheduleNextParallelization signature(self = "ParallelizeBackendOGS"): ...

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

ParallelizeBackend, ParallelizeBackendLocal, ParallelizeBackendSnow, ParallelizeBackendOGSremote

Examples

```
showClass("ParallelizeBackendOGS")
```

ParallelizeBackendOGSremote-class
Class "ParallelizeBackendOGSremote"

Description

Backend class supporting Open Grid Scheduler support on remote machines

Objects from the Class

Objects can be created by calls of the form .

Slots

`jids`: Object of class "ParallelizeBackendOGSstate" ~~

`config`: Object of class "list" ~~

`offline`: Object of class "logical" ~~

`signature`: Object of class "character" ~~

Extends

Class "ParallelizeBackend", directly.

Methods

getResult `signature(self = "ParallelizeBackendOGSremote")`: ...

initialize `signature(.Object = "ParallelizeBackendOGSremote")`: ...

initScheduling `signature(self = "ParallelizeBackendOGSremote")`: ...

lapply_dispatchFinalize `signature(self = "ParallelizeBackendOGSremote")`: ...

performParallelizationStep `signature(self = "ParallelizeBackendOGSremote")`: ...

pollParallelization `signature(self = "ParallelizeBackendOGSremote")`: ...

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

ParallelizeBackend, ParallelizeBackendLocal, ParallelizeBackendSnow, ParallelizeBackendOGSremote

Examples

```
showClass("ParallelizeBackendOGSremote")
```

ParallelizeBackendSnow-class
Class "ParallelizeBackendSnow"

Description

Backend class for parallelization on SNOW clusters

Objects from the Class

Objects can be created by calls of the form `new("ParallelizeBackendSnow", config, ...)`.

Slots

config: Object of class "list" ~~

offline: Object of class "logical" ~~

signature: Object of class "character" ~~

Extends

Class "ParallelizeBackend", directly.

Methods

initialize `signature(.Object = "ParallelizeBackendSnow")`: ...

lapply_dispatchFinalize `signature(self = "ParallelizeBackendSnow")`: ...

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

ParallelizeBackend, ParallelizeBackendLocal, ParallelizeBackendSnow, ParallelizeBackendOGSremote

Examples

```
showClass("ParallelizeBackendSnow")
```

parallelize_initialize *Initialize dynamic parallelization of ensuing parallelize calls*

Description

Initialzes the parallelization process. The config argument describes all parameters for as many backends as are available. Remaining arguments select a configuration for the ensuing parallelization from that description.

Usage

```
parallelize_initialize(Lapply_config = Lapply_config_default,
   stateClass = "LapplyState", backend = "local", freezerClass = "LapplyFreezer",
    ..., force_rerun = FALSE, sourceFiles = NULL, parallel_count = NULL)
```

Arguments

Lapply_config	A list describing possible configurations of the parallelization process. See Details.
stateClass	A class name representing parallelization states. Needs only be supplied if custom extensions have been made to the package.
backend	The name of the backend used. See Details and Examples.
freezerClass	The freezerClass used to store unevaluated calls that are to be executed in parallel. Needs only be supplied if custom extensions have been made to the package.
...	Extra arguments passed to the initializer of the stateClass.
force_rerun	So called offline computations are stateful. If a given rampUp has been completed an ensuing call - even a rerun of the script in a new R interpreter - reuses previous result. If set to TRUE force_rerun ignores previous results and recomputes the whole computation.
sourceFiles	Overwrite the sourceFiles entry in Lapply_config.
parallel_count	Overwrite the parallel_count entry in Lapply_config.

Details

config is a list with the following elements `config = list(max_depth = 5, parallel_count = 24, offline = TRUE );`

Value

Value NULL is returned.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

parallelize, parallelize_call

Examples

```
config = list(max_depth = 5, parallel_count = 24, offline = TRUE, backends = list(
  snow = list(
   localNodes = 1, splitN = 1, sourceFiles = c('RgenericAll.R', 'Rgenetics.R', 'RlabParallel.R')
  ),
  local = list(
    path = sprintf('%s/tmp/parallelize', tempdir())
  ),
  `ogs-1` = list(
    backend = 'OGS',
    freezerClass = 'LapplyPersistentFreezer',
    sourceFiles = c('RgenericAll.R', 'RlabParallel.R'),
    stateDir = sprintf('%s/tmp/remote', tempdir()),
    qsubOptions = sprintf('--queue all.q --logLevel %d', 2),
    doNotReschedulde = TRUE
  ),
  `ogs-2` = list(
    backend = 'OGS',
    freezerClass = 'LapplyPersistentFreezer',
    sourceFiles = c('RgenericAll.R', 'RlabParallel.R'),
    stateDir = sprintf('%s/tmp/remote', tempdir()),
    qsubOptions = sprintf('--queue subordinate.q --logLevel %d', 2),
    doSaveResult = TRUE
  ),
  `ogs-3` = list(
    backend = 'OGSremote',
    remote = 'user@localhost:tmp/remote/test',
    freezerClass = 'LapplyPersistentFreezer',
    sourceFiles = c('RgenericAll.R', 'RlabParallel.R'),
    stateDir = sprintf('%s/tmp/remote/test_local', tempdir()),
    qsubOptions = sprintf('--queue all.q --logLevel %d', 2),
    doSaveResult = TRUE
  )
));
# run ensuing parallelizations locally, ignore result produced earlier
parallelize_initialize(config, backend = "local", force_rerun = FALSE);
# run ensuing parallelizations on the snow cluster defined in the snow backend section
parallelize_initialize(config, backend = "local");
# run ensuing parallelizations on a local Open Grid Scheduler
parallelize_initialize(config, backend = "ogs-1");
# run same analysis as above with different scheduling options
parallelize_initialize(config, backend = "ogs-2");
# run same analysis on a remote Opend Grid Scheduler
# user 'user' on machine 'localhost' is used
parallelize_initialize(config, backend = "ogs-3");
```

parallelize_setEnable *Turn on/off the parallelization mechanism*

Description

This function changes the definition of Apply/Lapply/Sapply/parallelize/parallelize_call to turn the parallelization mechanism on or off.

Usage

```
parallelize_setEnable(state)
```

Arguments

```
state
```

Details

This function changes the definition of Apply/Lapply/Sapply/parallelize/parallelize_call to turn the parallelization mechanism on or off. The global environment is modified by a call to this function.

Value

The returned value is undefined.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

See Also

parallelize, parallelize_call, parallelize_initialize

Examples

```
parallelize_setEnable(FALSE);
Lapply
parallelize_setEnable(TRUE);
Lapply
```

readFile *Read content of file and return as character object.*

Description

Read content of file and return as character object.

Usage

```
readFile(path, prefixes = NULL, normalize = T, ssh = F)
```

Arguments

path	Path to the file to be read.
prefixes	Search for file by prepending character strings from prefixes.
normalize	Standardize pathes.
ssh	Allow pathes to remote files in scp notation.

Details

Read content of file and return as character object.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

Examples

```
parallel8 = function(e) log(1:e) %*% log(1:e);
cat(readFile(tempcodefile(parallel8)));
```

tempcodefile *Create a temporary file that contains the function definition of the argument.*

Description

Create a temporary file that contains the function definition of the argument so that this file can be sourced to re-instantiate the function.

Usage

```
tempcodefile(fcts)
```

Arguments

`fcts`

Details

Create a temporary file that contains the function definition of the argument so that this file can be sourced to re-instantiate the function. The temporary file is written to the temporary folder of the current R session.

Value

Returns the path to the file written.

Author(s)

Stefan Böhringer <r-packages@s-boehringer.org>

Examples

```
# code to be parallelized
parallel8 = function(e) log(1:e) %*% log(1:e);
parallel2 = function(e) rep(e, e) %*% 1:e * 1:e;
parallel1 = function(e) Lapply(rep(e, 15), parallel2);
parallel0 = function() {
  r = sapply(Lapply(1:50, parallel1),
    function(e)sum(as.vector(unlist(e))));
  r0 = Lapply(1:49, parallel8);
  r
}

codeFile = tempcodefile(c(parallel0, parallel1, parallel2, parallel8));
cat(readFile(codeFile));
```

Index