

Package ‘Rlsf’

April 17, 2009

Title Interface to the LSF Queuing System

Date 2007-04-13

Version 1.1.1

Author Chris Smith <csmith@platform.com>, Gregory Warnes <warnes@bst.rochester.edu>, Max Kuhn <max.kuhn@pfizer.com> and Nathan Coulter <nathan.coulter@pfizer.com>

Description This package provides functions for using R with the LSF cluster/grid queuing system.

Depends snow, gdata, chron

SystemRequirements Platform LSF development libraries

Maintainer Max Kuhn <max.kuhn@pfizer.com>

License LGPL

Repository CRAN

Date/Publication 2007-04-14 10:48:27

R topics documented:

bqueues	2
jobMonitor	2
lsf.control.job	4
lsf.ctrl	5
lsf.get.result	8
lsf.job.status	9
lsf.numcpu	10
lsf.parRapply	11
lsf.run.job	13
lsf.submit	14
lsf.submit2	16
lsfTmpDir	17
scat	18

Index	19
--------------	-----------

 bqueues

Displays information about queues

Description

This function parses the command line output of the bqueues command line statement

Usage

```
bqueues ()
```

Details

a system call is used to collect the bqueues output and this is imported into R

Value

a data frame where each row is a queue. There are columns for the number of jobs suspended, pending, running etc. There is also a column, `ratio`, that is the logit of the ratio of non-active jobs (pending or suspended) out of all the jobs.

Author(s)

Max Kuhn

 jobMonitor

Management of one or more Rlsf jobs

Description

This function will monitor and manage jobs submitted using `lsf.submit` or `lsf.submit2`.

Usage

```
jobMonitor(x, pause = 1, timeLimit = TRUE, buffer = 20, verbose = TRUE)
```

Arguments

<code>x</code>	a list. For one job, it can be the object produced by <code>lsf.submit</code> or <code>lsf.submit2</code> . When there are multiple jobs, <code>x</code> should be a list of objects resulting from either of these two functions
<code>pause</code>	the number of seconds to pause before checking the results again
<code>timeLimit</code>	a logical: should the function kill jobs that are taking an inordinate amount of time? (see details below)
<code>buffer</code>	a multiplier to create a failsafe time when <code>timeLimit = TRUE</code> .
<code>verbose</code>	a logical: should updates be printed?

Details

This function will check to see if jobs are completed (either with return codes of "DONE" or "EXIT"). Once a job is finished, the results are immediately obtained using `lsf.get.result`. If `x` contains more than one job, the results of all of the jobs are returned in a list; otherwise the results are returned directly.

The option `timeLimit = TRUE` is useful for cases where very similar jobs are being run, such as bootstrapping or cross-validation. In this case, the monitor will wait until half of the jobs are finished and when will estimate a "failsafe" for the remaining jobs. the failsafe takes the maximum execution time of the finished jobs and multiplies it by `buffer`. If the elapsed time for any of the remaining jobs exceeds this limit, they are killed.

If all of the jobs are done before `jobMonitor` is called, the failsafe is set to 0.1 (days).

Value

When `x` contains more than one call to `lsf.submit` or `lsf.submit2` the output is a list of the results for each job; otherwise it is the result of the single job.

If the job is killed by `jobMonitor`, it will return the string "killed due to excessive run time"

Author(s)

Max Kuhn

See Also

[lsf.submit](#) and [lsf.submit2](#)

Examples

```
test <- function(slowDown = FALSE)
{
  # add this or the jobs will be done before jobMonitor gets them
  Sys.sleep(15)

  out <- Sys.info()["version"]

  # test the failsafe by making the job run really long
  if(slowDown) Sys.sleep(600)
  out
}

numJobs <- 20
jobData <- vector(mode = "list", length = numJobs)
for(i in 2:numJobs) jobData[[i]] <- lsf.submit2(test)
jobData[[1]] <- lsf.submit2(test, slowDown = TRUE)

# make the cutoff short for demo purposes
versionNames <- jobMonitor(jobData, buffer = 1)
```

lsf.control.job *Control a job that was submitted to LSF*

Description

Manage a job which has been previously submitted to LSF using the `lsf.submit` function. Control actions include the termination of the job, the suspension of a running job, and the resumption of a suspended job.

Usage

```
lsf.kill.job(job)
lsf.suspend.job(job)
lsf.resume.job(job)
```

Arguments

`job` The list returned from a previous call to `lsf.submit`

Details

`lsf.kill.job` will ask LSF to terminate a job that was previously submitted using `lsf.submit`.

`lsf.suspend.job` will ask LSF to suspend a job (i.e. send it a stop signal) that was previously submitted using `lsf.submit`.

`lsf.resume.job` will ask LSF to resume a job that had previously been suspended using the `lsf.suspend.job` function.

Value

Returns 0 if the control request was sent successfully, and -1 if there was an error sending the request.

Note

When LSF control functions are called, the call represents a request to take the action. Depending on the configuration of LSF, the actual action taken is not pre-defined. For example, LSF can be configured to send a different signal from the default SIGSTOP in order to suspend a job. If this different signal is sent by LSF, but the job ignores the signal, then the job is not actually stopped. There are similar configuration overrides for both termination and resumption of jobs.

Author(s)

Chris Smith <csmith@platform.com>

See Also

[lsf.submit](#)

Examples

```

# define the function to remotely run
myfunc <- function() { 2+3 }

# submit the function to run in batch
job <- lsf.submit(myfunc)

# check if the job is in the "RUN" state
lsf.job.status(job)

# when job is in RUN state, suspend it
lsf.suspend.job(job)

# check to see if the job is in "USUSP" state
lsf.job.status(job)

# when job is in "USUSP" state, resume it
lsf.resume.job(job)

# check that job is back in the "RUN" state.
# Might take a minute or two for signal to propagate.
lsf.job.status(job)

# when job is running, kill it
lsf.kill.job(job)

# the job should be in "DONE" or "EXIT" state
lsf.job.status(job)

```

lsf.ctrl

Control over job parameters

Description

This function sets many, many arguments for LSF jobs.

Usage

```

lsf.ctrl(
  savelist = c(), packages = NULL, dbg = FALSE, env = sys.frame(sys.parent()),
  tmpPath = getwd(), jobName = NULL, queue = NULL, askedHosts = NULL,
  resReq = NULL, rlimit_cpu = NULL, rlimit_fsize = NULL, rlimit_data = NULL,
  rlimit_stack = NULL, rlimit_core = NULL, rlimit_rss = NULL,
  rlimit_nofile = NULL, rlimit_open_max = NULL, rlimit_swap = NULL,
  rlimit_run = NULL, rlimit_process = NULL, hostSpec = NULL, numProcessors = 1,
  dependCond = NULL, beginTime = NULL, termTime = NULL, sigValue = NULL,
  inFile = NULL, outFile = "/dev/null", errFile = NULL, command = NULL,

```

```

chkpntPeriod = NULL, chkpntDir = NULL, nxf = NULL, xFile = NULL,
preExecCmd = NULL, mailUser = NULL, delOptions = 0, projectName = NULL,
maxNumProcessors = 1, loginShell = NULL, userGroup = NULL, exceptList = NULL,
exclusive = NULL, notifyBegin = NULL, notifyEnd = NULL, restart = NULL,
restartForce = NULL, rerunnable = NULL, chkpnt_copy = NULL,
chkpnt_force = NULL, interactive = NULL, pty = NULL, pty_shell = NULL,
hold = NULL, R = "R")

```

Arguments

savelist	character vector giving the names of local objects that should be copied to each worker process before computation is started.
packages	List of library packages to be loaded by each worker process before computation is started.
dbg	Nathan
env	the environment to get teh R objects from
tmpPath	a path where temporary files are stored
jobName	a label for the jobs (-J)
queue	the job queue (-q). fo Pfizer, either "immed_single", "immed_multi", "overnight_single", "overnight_multi" or "asavailable"
askedHosts	the machine name -m
resReq	a resource string -R
rlimit_cpu	see LSF documentation
rlimit_fsize	see LSF documentation
rlimit_data	see LSF documentation
rlimit_stack	the service class where the job is run -S
rlimit_core	see LSF documentation
rlimit_rss	resident memory size, in kilobytes. see LSF documentation on -M
rlimit_nofile	see LSF documentation
rlimit_open_max	see LSF documentation
rlimit_swap	the virtual memory swap limit (in KB) -v
rlimit_run	see LSF documentation
rlimit_process	the limit on the number of processes -p
hostSpec	the total CPU time for the job -c
numProcessors	see LSF documentation
dependCond	a dependency condition for starting the job -w
beginTime	the start time for the job in seconds since 00:00:00 GMT, Jan. 1, 1970 -b
termTime	the termination deadline for the job in seconds since 00:00:00 GMT, Jan. 1, 1970 -t

sigValue	see LSF documentation
inFile	a file to get standard input from <code>-i</code>
outFile	a file to append standard out to <code>-o</code>
errFile	a file to append standard error output to <code>-e</code>
command	see LSF documentation
chkpntPeriod	enables checkpoints for a period of time <code>-k</code>
chkpntDir	a directory path for checkpoints <code>-k</code>
nxf	a file to copy between th submission host and the execution server <code>-f</code>
xFile	a file to copy between th submission host and the execution server <code>-f</code>
preExecCmd	a command to execute prior to running the remote job <code>-E</code>
mailUser	see LSF documentation
delOptions	see LSF documentation
projectName	assigns the job to the specified project <code>-P</code>
maxNumProcessors	the maximum number of processors to run the job
loginShell	initializes the execution environment using a loginf shell <code>-L</code>
userGroup	a user group to associate the job with <code>-G</code>
exceptList	see LSF documentation
exclusive	exclusive execution mode <code>-x</code>
notifyBegin	an email address to send an email to once the process is started; see LSF documentation on <code>-B</code>
notifyEnd	an email address to send an email to once the process is finished; see LSF documentation on <code>-N</code>
restart	see LSF documentation
restartForce	see LSF documentation
rerunnable	make the job re-runnable <code>-r</code>
chkpnt_copy	see LSF documentation on <code>lsf_submit(3)</code>
chkpnt_force	see LSF documentation on <code>lsf_submit(3)</code>
interactive	see LSF documentation on <code>-I</code> , requires "interactive" argument
pty	see LSF documentation on <code>-Ip</code> , requires "pty" and "interactive"
pty_shell	interactive mode; new jobs cannot be submitted until the current oe is finished <code>-I</code>
hold	suspend the job in PSUP mode <code>-H</code>
R	the command-line call to R (could be an alias or a path)

Value

a list or arguments specified by the function call or the defaults

Author(s)

Nathan coulter

References

<http://www.platform.com/Products/Platform.LSF.Family/Platform.LSF/Home.htm>

lsf.get.result

Get the result of a function run through LSF

Description

Retrieve the function result from the asynchronous execution of an R function through LSF.

Usage

```
lsf.get.result(job)
```

Arguments

job The list returned from a previous call to `lsf.submit`

Details

This function will retrieve the result of the function provided as an argument to `lsf.submit`. Since `lsf.submit` runs asynchronously, the result must be explicitly requested.

Value

This function returns the result of the remote invocation of the `func` argument to the `lsf.submit` call, or will return NULL if there is no result available at the time of the call (the result can only be retrieved when `lsf.job.status` returns "DONE").

The function is invoked remotely using the `try` function. If there is an error invoking the submitted function, an object of class "try-error" containing the error message will be returned.

Author(s)

Chris Smith <csmith@platform.com>

See Also

[lsf.submit](#), [lsf.job.status](#), [try](#)

Examples

```
# define the function to remotely run
myfunc <- function() { 2+3 }

# submit the function to run in batch
job <- lsf.submit(myfunc)

# check if the job is finished yet
lsf.job.status(job)

# retrieve the result of myfunc
# should only be called when lsf.job.status returns 'DONE'
res <- lsf.get.result(job)
```

lsf.job.status	<i>Check the status of a function evaluated through LSF.</i>
----------------	--

Description

Check the status of a function evaluated through LSF.

Usage

```
lsf.job.status(job)
```

Arguments

job The list returned from a previous call to `lsf.submit`

Details

`lsf.job.status` will return a string representing the state of a function evaluated through LSF (from a previous call to `lsf.submit`).

Value

Returns a string representation of the state of the LSF job (e.g. "PEND", "RUN", "DONE"). The possible states returned from `lsf.job.status` are described in the manual page for the LSF `bjobs` command.

Author(s)

Chris Smith <csmith@platform.com>

See Also

[lsf.submit](#)

Examples

```
# define the function to remotely run
myfunc <- function() { 2+3 }

# submit the function to run in batch
job <- lsf.submit(myfunc)

# check for the status of the job
lsf.job.status(job)
```

`lsf.numcpu`*Determine the number of CPUs assigned by LSF*

Description

Determine the number of CPUs assigned by LSF.

Usage

```
lsf.numcpu()
```

Arguments

None.

Details

This function determines the number of hosts present in the `LSB_HOSTS` environment variable. If the environment variable does not exist or is empty, an error will be generated.

Value

The number of CPUs assigned the R process by LSF.

Author(s)

Gregory R. Warnes <warnes@bst.rochester.edu>

Examples

```
# get the number of cpus, fail if unable to do so
lsf.numcpu()
```

lsf.parRapply	<i>Apply a function to each row/column of a matrix, using Rlsf for parallel processing</i>
---------------	--

Description

Apply a function to each row/column of a matrix, using Rlsf for parallel processing

Usage

```
lsf.parRapply(x, fun, ..., join.method=cbind, njobs,
             batch.size=getOption('lsf.block.size'), trace=TRUE,
             packages=NULL, savelist=NULL)
```

```
lsf.parCapply(x, ...)
```

```
lsf.apply.model(fun, matrix, ..., njobs,
               batch.size=getOption('lsf.block.size'),
               packages = .packages(), savelist = NULL)
```

Arguments

<code>x, matrix</code>	Data matrix
<code>fun</code>	Function to apply to each row of <code>x</code>
<code>...</code>	Additional arguments to <code>fun</code>
<code>join.method</code>	Function used to join together results from each parallel job. **Note: When <code>fun</code> returns a vector, this needs to be a function that operates on the transpose of the matrix.
<code>njobs</code>	Number of parallel jobs to use.
<code>batch.size</code>	Number of rows to include in each parallel job. Defaults to the value of <code>getOption('lsf.block.size')</code> .
<code>trace</code>	Show progress of computational jobs.
<code>packages</code>	List of library packages to be loaded by each worker process before computation is started.
<code>savelist</code>	Character vector giving the names of local objects that should be copied to each worker process before computation is started.

Details

`lsf.parRapply` applies the supplied function to each row of the supplied matrix `x`. `lsf.parCapply` does the same for columns. `lsf.apply.model` is a wrapper around `lsf.parRapply` that does some extra error checking, has better handling of warnings, and transposes the output to match users's expectations.

The arguments `batch.size` and `njobs` are mutually exclusive. If `batch.size` is set, the data matrix `x` into sections of size `batch.size`. The number of sections will determine the number of jobs.

On the other hand, if `numjobs` is set, the data matrix `x` will be divided into `numjobs` components of near equal size.

The arguments `packages` and `savelist` can be used to properly initialize the worker processes.

Value

A matrix, vector, or list (depending on the value of `'join.method'`) containing the result of applying `fun` to each row of the matrix `x`. As each call to `fun` is wrapped in a `try` statement, the returned object may contain `try-error` objects.

Author(s)

Gregory R. Warnes (warnes@bst.rochester.edu)

See Also

[lsf.apply.model](#), [lsf.submit](#)

Examples

```
#####
## Example of lsf.parRapply
#####
set.seed(12345)
x <- matrix(rnorm(1e6+20), ncol=20)
dim(x)
means <- lsf.parRapply(x, mean, na.rm=TRUE, join.method=c, njobs=20)
means <- lsf.parRapply(x, mean, na.rm=TRUE, join.method=c, batch.size=1000)

####
## Example of lsf.apply.model
####
library(Rlsf)
library(gdata);

# generate an example data set
set.seed(1)
age <- rnorm(200, 40, 12)
sex <- factor(sample(c('female', 'male'), 200, TRUE))
logit <- (sex=='male') + (age-40)/5
y <- ifelse(runif(200) <= plogis(logit), 1, 0)
dataframe <- data.frame(age=age, sex=sex)

# create a matrix made up of rows containing the y data
ymat <- matrix(y, nrow=30127, ncol=length(y), byrow=TRUE)

# define the model fitting function
```

```

fun <- function(y, covariates)
{
  fit <- lm( y ~ age*sex, data=covariates )
  sum <- summary(fit)
  retval <- unmatrix( coef(sum) )
  retval
}

# now fit the model to each row (will yield identical results for each...)
ret <- lsf.apply.model (
  fun,
  matrix = ymat,
  covariates = dataframe,
  savelist="unmatrix",
  batch.size=3000
)

```

lsf.run.job

*Evaluate a function synchronously through LSF***Description**

Evaluates an R function synchronously through the LSF queuing system.

Usage

```
lsf.run.job(func, ..., savelist = c(), packages=NULL, ncpus = 1, debug =
FALSE, interval = 15)
```

Arguments

func	This argument provides the name of the function that will be evaluated through LSF. It must be defined in the current scope.
...	Any arguments to func should be passed after the function name. They must be within the current scope, and must be provided to the remote function call as part of the savelist
savelist	Character vector giving the names of local objects that should be copied to each worker process before computation is started.
packages	List of library packages to be loaded by each worker process before computation is started.
ncpus	A number indicating how many cpus should be allocated to the function call. Used in conjunction with the snow and Rmpi packages (see notes).
debug	An argument which indicates whether function debugging should be turned on.
interval	Defines the amount of time (in seconds) to wait between polls for the job status. If this is too low, it will cause undue load on the LSF scheduler.

Details

`lsf.run.job` is a convenience function that provides the ability to synchronously evaluate an R function through LSF. It is implemented by composing the `lsf.submit`, `lsf.job.status`, and `lsf.get.result` R functions.

Value

Returns the result of `func` as evaluated through LSF.

Author(s)

Chris Smith <csmith@platform.com>

See Also

[lsf.submit](#), [lsf.job.status](#), [lsf.get.result](#)

Examples

```
# define the function to remotely run
myfunc <- function() { 2+3 }

# submit the function to run in batch
result <- lsf.run.job(myfunc)
```

`lsf.submit`

Submit an R function to run through LSF

Description

Submit an R function to run in batch through the LSF distributed queuing environment.

Usage

```
lsf.submit(func, ..., savelist = c(), packages = NULL, ncpus = 1,
           debug = FALSE)
```

Arguments

<code>func</code>	This argument provides the name of the function that will be run through LSF. It must be defined within the current scope.
<code>...</code>	Any arguments to <code>func</code> should be passed after the function name. They must be within the current scope, and must be provided to the remote function call as part of the <code>savelist</code>
<code>savelist</code>	Character vector giving the names of local objects that should be copied to each worker process before computation is started.

packages	List of library packages to be loaded by each worker process before computation is started.
ncpus	A number indicating how many cpus should be allocated to the function call. Used in conjunction with the snow and Rmpi packages (see notes).
debug	An argument which indicates whether function debugging should be turned on.

Details

This function takes an R function plus arguments, and submits the function to run as a batch job within the LSF distributed queuing environment.

The function will store the environment specified in `savelist` within a temporary Rdata file, which will then be used on a remote host to restore the R environment for the function call.

Since `func` is run asynchronously, the result of the `lsf.submit` is not the result of `func`. In order to retrieve the result, the `lsf.get.result` function must be used. If synchronous execution is desired, use the `lsf.run.job` function.

By default, R is invoked as "R". If the environment variable "RLSF_R" is set, its value is used to invoke R.

Value

jobid	The job identifier returned from LSF.
fname	The file name of the environment passed to the remote function call.
debug	A boolean flag indicating whether debugging should be turned on or not. If debugging is turned on, then the standard output and standard error of the remote R process invocation will be placed in a file named <code>Rlsf_job_output.<jobid></code> , where <code>jobid</code> is the same as the LSF job id returned by <code>lsf.submit</code>

Note

When submitting parallel jobs (i.e. `ncpus` greater than 1), the `Rlsf` package will utilize the `snow` and `Rmpi` packages for managing the parallel job. At this time, no other parallel packages are supported (e.g. `snow` with `PVM`).

Author(s)

Chris Smith <csmith@platform.com>

See Also

[lsf.job.status](#), [lsf.get.result](#), [lsf.run.job](#)

Examples

```
# define some variables
a <- 2
b <- 3
```

```
# define the function to remotely run
myfunc <- function(x,y) { x + y }

# submit the function to run in batch
job <- lsf.submit(myfunc, a, b, savelist=c("a", "b"))
```

lsf.submit2

Submit an R function to run through LSF (remix)

Description

Submit an R function to run in batch through the LSF distributed queuing environment.

Usage

```
lsf.submit2(func, ctrl=lsf.ctrl(), ...)
```

Arguments

<code>func</code>	This argument provides the name of the function that will be run through LSF. It must be defined within the current scope.
<code>...</code>	Any arguments to <code>func</code> should be passed after the function name. They must be within the current scope, and must be provided to the remote function call as part of the <code>savelist</code>
<code>ctrl</code>	a list of options. See lsf.ctrl

Details

This function has the same purpose as [lsf.submit](#) but it has several differences.

First, the objects that are do be passed to teh function are copied to the fuction's workspace. This adds some overhead, but makes the function more modular and minimizes the scoping complexity (and thus shortens coding time).

Second, many of the LSF command line options are now exposed through the control object (see [lsf.ctrl](#) for more details). Now, the user can specify specific isntalls of R to use, which queues to send the job to or a resource string that can taget one or more machines.

Value

<code>jobid</code>	The job identifier returned from LSF.
<code>fname</code>	The file name of the environment passed to the remote function call.
<code>debug</code>	A boolean flag indicating whether debugging should be turned on or not. If debugging is turned on, then the standard output and standard error of the remote R process invocation will be placed in a file named <code>Rlsf_job_output.<jobid></code> , where <code>jobid</code> is the same as the LSF job id returned by <code>lsf.submit</code>

Note

When submitting parallel jobs (i.e. `ncpus` greater than 1), the `Rlsf` package will utilize the `snow` and `Rmpi` packages for managing the parallel job. At this time, no other parallel packages are supported (e.g. `snow` with `PVM`).

Author(s)

Chris Smith <csmith@platform.com>, Max Kuhn <max.kuhn@pfizer.com> and Nathan Coulter <nathan.coulter@pfizer.com>

See Also

[lsf.job.status](#), [lsf.get.result](#), [lsf.run.job](#)

Examples

```
data(iris)
library(MASS)

ldaJob1 <- lsf.submit2(
  func = lda,
  formula = as.formula(Species ~ .),
  data = iris,
  ctrl = lsf.ctrl(
    packages = "MASS",
    savelist = "iris",
    queue = "overnight_single"))

ldaFit <- jobMonitor(ldaJob1)
```

lsfTmpDir

Rlsf working directory

Description

Create a working directory for `Rlsf`

Usage

```
lsfTmpDir()
```

Details

This function determines the temp directory and creates a subdirectory. If a subdirectory cannot be created, an error is thrown.

Value

a character string containing the path

Author(s)

Max Kuhn

scat

Display debugging text

Description

If `getOption('DEBUG')==TRUE`, write text to `STDOUT` and flush so that the text is immediately displayed. Otherwise, do nothing.

Usage

```
scat(...)
```

Arguments

... Arguments passed to `cat`

Value

NULL (invisibly)

Author(s)

Gregory R. Warnes (warnes@bst.rochester.edu)

See Also

[cat](#)

Examples

```
options(DEBUG=NULL) # makee sure DEBUG isn't set
scat("Not displayed")
```

```
options(DEBUG=TRUE)
scat("This will be displayed immediately (even in R BATCH output \n")
scat("files), provided options()$DEBUG is TRUE.")
```

Index

- *Topic **print**
 - scat, [18](#)
- *Topic **utilities**
 - bqueues, [1](#)
 - jobMonitor, [2](#)
 - lsf.control.job, [3](#)
 - lsf.ctrl, [5](#)
 - lsf.get.result, [8](#)
 - lsf.job.status, [9](#)
 - lsf.numcpu, [10](#)
 - lsf.parRapply, [11](#)
 - lsf.run.job, [13](#)
 - lsf.submit, [14](#)
 - lsf.submit2, [16](#)
 - lsfTmpDir, [17](#)
- bqueues, [1](#)
- cat, [18](#)
- jobMonitor, [2](#)
- lsf.apply.model, [12](#)
- lsf.apply.model(*lsf.parRapply*),
[11](#)
- lsf.control.job, [3](#)
- lsf.ctrl, [5, 16](#)
- lsf.get.result, [2, 8, 14, 15, 17](#)
- lsf.job.status, [8, 9, 14, 15, 17](#)
- lsf.kill.job(*lsf.control.job*), [3](#)
- lsf.numcpu, [10](#)
- lsf.parCapply(*lsf.parRapply*), [11](#)
- lsf.parRapply, [11](#)
- lsf.resume.job(*lsf.control.job*),
[3](#)
- lsf.run.job, [13, 15, 17](#)
- lsf.submit, [2–4, 8, 9, 12, 14, 14, 16](#)
- lsf.submit2, [2, 3, 16](#)
- lsf.suspend.job
(*lsf.control.job*), [3](#)
- lsfTmpDir, [17](#)
- scat, [18](#)
- try, [8](#)