Package 'supr2'

March 21, 2020

Version 3.6.2

Date 2020-03-01

Title Experimenting Parallel and Distributed Computation in R

Author Chuanhai Liu

Maintainer Chuanhai <chuanhai@purdue.edu>

Description Support for parallel distributed computation, including multithreading, distributed file and memory management, and flexible cluster-wise iterative computing.

License GPL (>=2)

Imports stats, graphics

Suggests methods

NeedsCompilation yes

Depends R (>= 3.5.0)

R topics documented:

supr2-package *SupR: an experimental R package for parallel and distributed computing*

Description

Support for parallel distributed computation, including multithreading, distributed file and memory management, and flexible cluster-wise iterative computing.

Details

SupR consists of six main components:

1. *SuprContext*: a cluster of distributed context that is based on a collection of stand-alone programs, namely, driver, master, worker, taskrunner. These programs, including those for DFS (dfs_name to run as a namenode, and dfs_data to run datanodes), are available in the package's bin directory. The main user interface is provided as the *SuprContext* function;
2. *Thread*: a set of functions for multithreading, reflecting a collection of experimental efforts to make use of multithreading in R without any change of the R system code (>= R-3.6.1, at the developmental time).
3. *SuprMR*: a set of functions for implementing SupR's method for cluster-wide iterative statistical computing on distributed data, which is traditionally known as map-and-reduce or map-and-combine.
4. *DCL*: a simple implementation of data change listener.
5. *SDFS*: a distributed file system, or more generally, a distributed object system. The user interface R functions to access DFS have names starting with *DD.*.
6. *Rython*: a collection of R and Python functions for interface to Python https://www.python.org/.

Author(s)

Chuanhai Liu

Maintainer: Chuanhai Liu

References

http://www.stat.purdue.edu/~chuanha

See Also

DCL DD Thread Rython SuprMR SuprContext SuprJob Sync.eval, Sync.wait

Examples

```
library(supr2)
## Not run: SuprContext()
## Not run: SuprContext(shutdown)
## Not run: DD.list()
## Not run: Thread.join(thread)
## Not run: thread <- Thread.new({1+2})
## Not run: Thread.join(thread)
```

`Cluster` *SupR Cluster Interface*

Description

Functions to interface with the SupR cluster system. These incldue those that can obtained by `grep("job|[Cc]luster",ls("package:supr2"),value=TRUE)`.

Usage

```
ClusterEnv

cluster.eval(expr, data, envir = NULL, collect = FALSE, dcl = NULL,
    dcl.expr = NULL, dcl.envir = NULL, env = parent.frame(),
    wait = TRUE, ...)
cluster.get(uri, name, remove = FALSE)
cluster.put(name, ...)

job.get(job_id, timeout = 60)
job.isDone(job_id)
job.cancel(job_id)
job.info(job_id)
job.rm(job_id)

collect(x)
cluster.combine(x, combine, bykey = FALSE, env = parent.frame())

nextEvent(...)
cond.count(..., cond.name = "default", env = parent.frame())
```

Arguments

`expr`	a meaningful expression executable by `Taskrunners`.
`data`	either a list object of subsets or a distributed data in `SDFS`.
`envir`	an environment objection used by `Taskrunners` to to evaluate `expr` for subsets. See `Details`.
`env`	CHANGE IT TO parent.env? TODO

collect	a logical object. See Details.
dcl	a data-change listener. See Details.
dcl.expr	the expression to run by dcl. See Details.
uri	a URI object. See Details.
name	a character string used as a name.
job_id	an integer used as job id.
timeout	an amount of time to wait for the job to finish.
remove	a logical object. FIXME...
...	optional arguments.

Details

...

Value

cluster.eval returns the evaluated value for wait = TRUE, otherwise, it returns an integer as the job id. The returned result is a list of TaskrunnerValue objects from individual taskrunners. A tr.val (Taskrunner Value) object is a list including a *value* component, a *warnings* component, and *taskrunner* name. IN case of any errors, the *value* is replaced by an *error* component. See Examples.

Note

...

See Also

sync.eval, sync.wait, sync.notify, ClusterEnv, JobEnv, SharedEnv

Examples

```
## Not run:
  SuprContext()

## End(Not run)
# An echo example
## Not run:
  expr.echo <- quote({
        subsets <- list()
        while(!is.null(x <- nextSubset())) {
             subsets <- append(subsets, x)
             Sys.sleep(1)
        }
        subsets
  })

  results <- cluster.eval(expr.echo, as.list(1:100))
```

```
# The number of taskrunners for completing all the tasks
  length(results)
# Check if there are any errors occurred
  err <- unlist(lapply(results, `[`, "error"))
  if(!is.null(err)) err
# The number of tasks completed individual taskrunners
  names(results) <- unlist(lapply(results, `[[`, "taskrunner"))
  lapply(lapply(results, `[[`, "value"), length)
# Task results
  structure(unlist(lapply(results, `[`, "value")), names = NULL)


## End(Not run)

# A cluster-level combine example
## Not run:
  expr.combine <- quote({
    my.combine <- function(x, y){x+y}
    x <- nextSubset()
    while(!is.null( y <- nextSubset() )){
           x <- my.combine(x, y)
           Sys.sleep(1)
    }
    cluster.combine(x, my.combine)
  })

  results <- cluster.eval(expr.combine, data=as.list(1:100))

  err <- unlist(lapply(results, `[[`, "error"))
  if(!is.null(err)) err

  unlist(lapply(results, `[[`, "value"))

## End(Not run)

# A cluster-level combine_by_key example

## Not run:
  SuprContext()

  expr.cbk <- quote({
    Sys.sleep(1)
    x <- nextSubset()
    while(!is.null(y <- nextSubset())){
           x <- x + y
    }

    data <- new.env(parent=emptyenv())

    if(x > 26) x = 26
    keys <- sample(letters, x)
    for(i in 1:x){
      assign(keys[i], x, data)
```

```
    }
    combine <- function(x, y){x+y}
    cluster.combine(data, combine, bykey = TRUE)
  })

  results <- cluster.eval(expr.cbk, data=as.list(1:100))

  err <- unlist(lapply(results, `[[`, "error"))
  if(!is.null(err)) err

  collect(results)


## End(Not run)

# A simple EM example using the standard synchronization method.
# For the DD object EM_data, see the DD example
## Not run:
  expr.em <- quote({ subsets <- list()
    while(!is.null(sub <- nextSubset())) { x <- sub$x[,1]
      subsets <- append(subsets, list(x)); Sys.sleep(0.1)
    }
    n.subsets = length(subsets)
    if(n.subsets==0){ stop("FIXME"); X <- numeric(0); mu <- 0 } else {
      X <- unlist(subsets); mu <- mean(X[!is.na(X)])
    } # print(head(X)); print(typeof(X)); print(class(X))

    mis <- is.na(X); n.com <- length(X)
    E.step <- function(mu){
      X[mis] <- mu
      list(sx = sum(X), n = n.com)
    }
    my.combine <- function(x, y){ list(sx = x$sx + y$sx, n = x$n + y$n) }
    M.step <- function(SS){ SS$sx/SS$n }

    n <- 0; max.n = 1000; this.job <- paste0("JOB-", .job_id)
    while(!is.null(mu)) { n <- n  + 1; mu0 <- mu
      SS <- E.step(mu)
      my.result <- cluster.combine(SS, my.combine)
      if (is.null(my.result)) {
        send.driver(paste("n.iterates =", n, "Waiting..."))
        mu <- sync.eval(JobEnv$std_EM, {
  sync.wait(JobEnv$std_EM)
  JobEnv$mu
          })
      } else {
        mu <- if(n >= max.n) NULL else  M.step(my.result)
        send.driver(paste("n.iterates =", n, "mu = ", mu))
JobEnv$mu <- mu
        sync.eval(JobEnv$std_EM, sync.notify(JobEnv$std_EM, all=TRUE) )
      }
    }
    list(mu=mu0, Sx=SS$sx, n = n, n.mis=sum(mis), n.subsets=n.subsets)
```

```
  })

  result <- cluster.eval(expr.em, data=DD("EM_data"))

## End(Not run)

# A simple EM example using the countdown variables.
# For the DD object EM_data, see the DD example
## Not run:
   expr.em <- quote({ subsets <- list()
    while(!is.null(sub <- nextSubset())) { x <- sub$x[,1]
      subsets <- append(subsets, list(x)); Sys.sleep(0.1)
    }
    n.subsets = length(subsets)
    if(n.subsets==0){ stop("FIXME"); X <- numeric(0); mu <- 0 } else {
      X <- unlist(subsets); mu <- mean(X[!is.na(X)])
    } # print(head(X)); print(typeof(X)); print(class(X))

    mis <- is.na(X); n.com <- length(X)
    E.step <- function(mu){
      X[mis] <- mu
      list(sx = sum(X), n = n.com)
    }
    combine <- function(x, y){ list(sx = x$sx + y$sx, n = x$n + y$n) }
    M.step <- function(SS){ SS$sx/SS$n }

    # Starting values
    this.job <- paste0("JOB-", .job_id)
    n.iter <- 0; if(!exists("max.n.iter")) max.n.iter = 1000
    if(exists("mu0")) mu <- mu0

    while(!is.null(mu)) { n.iter <- n.iter  + 1; mu0 <- mu
      SS <- E.step(mu)
      my.result <- cluster.combine(SS, combine)
      if (is.null(my.result)) {
        mu <- cond.count()$theta
      } else {
        mu <- if(n.iter >=  max.n.iter) NULL else M.step(my.result)
        send.driver(paste("n.iterates =", n.iter, "mu:", mu))
        mu <- cond.count(theta=mu)$theta
      }
    }
    list(mu=mu0, Sx=SS$sx, n.com = n.com, n.mis=sum(mis), n.subsets=n.subsets)
  })

  result <- cluster.eval(expr.em, data=DD("EM_data"))

## End(Not run)

# Add a data change listener in the last EM example.
# For the DD object EM_data, see the DD example
## Not run:  # .Last <- function() SuprContext(shutdown)
  my.dcl.expr <- quote({ # Sys.sleep(30)
```

```
    #
    #thread.openXTerm()
    sink("/dev/pts/29")
    thread.startInputHandler()
    mu <- numeric(0)
    while(!is.null(e <- nextEvent())){
      cond <- e$condition
      cond <- cond[!unlist(lapply(cond, is.null))][[1]]
      value <- cond$theta
      print(value)
      mu <- c(mu, value)

      s <- ((length(mu)+1L)%/%2L) : length(mu)
      plot(mu[s])
      if(length(mu)>10 && max(abs(diff(mu[s])))<10E-10){
#        e <- nextEvent(value=NULL) # FIXME
#             stop("Converged??")
      }
    }
    print("DONE!")
    Sys.sleep(10)
  })

  env <- new.env(parent = emptyenv())
  env$max.n.iter = 100
  env$mu0 = 10

  res <- cluster.eval(expr.em, data=DD("EM_data"), envir = env,
            dcl="my.dcl", dcl.expr=my.dcl.expr, dcl.env = new.env())



## End(Not run)
```

DD *SupR Distributed Data System*

Description

Functions to access the SupR distributed data system. These incldue those that can obtained by `grep("dd|DD",ls("package:supr2"),value=TRUE)`.

Usage

```
DD(name)
DD.close(name, save = TRUE)
DD.exists(name, ...)
```

```
DD.get(name, subset.names, ..., env = parent.frame())
DD.list(name = ".")
DD.mv(src, dest)
DD.open(name)
DD.options(...)
DD.persist(name, level = c("file", "tmp", "shm", "mem", "null"))
DD.put(dd.name, subsets, subset.names)
DD.replicate(name)
DD.rm(name, recursive = FALSE)
```

Arguments

`name, dd.name`	a DD name, a character string representing a Unix file path.
`subset.names`	a character vector of Unix simple file names.
`env`	an environment for evalutions of arguments.
`src, dest`	DD names.
`level`	one of the values given in its default.
`subsets`	objects representing subsets.
`subset.names`	a character vector specifying the subset names.
`...`	optional arguments.

Details

...

Value

...

Note

...

See Also

Monitor, Concurrency.

Examples

```
# Start a SupR context with the default configuration
## Not run:  SuprContext()


# Show the DFS_namenode's connections
## Not run:  SuprContext(dfs)
```

```
# Create a distributed data, named "EM_data", consisting of
# simulated subsets, and upload it into DFS
## Not run:  data.name = "EM_data"
  if(DD.exists(data.name)) DD.rm(data.name, TRUE)

  DD.open(data.name)

  N <- 100L  # number of subsets
  K <- 1000L # subset size
  mis.prob <- 0.5  # fraction of missing values

  if(!dir.exists(data.name))
    dir.create(data.name)

  files <- paste0(data.name, "/part_", 1:N, ".rdata")
  for(i in 1:length(files)){
    x <- rnorm(K)
    mis <- runif(K) < mis.prob
    x[mis] <- NA
    x <- data.frame(x=x)
    save(N, x, file=files[i])
  }

  DD.put(data.name, files, NULL)
  DD.list(data.name)

  #unlink(data.name,  recursive = TRUE, force = TRUE)

## End(Not run)

## Create one more replicate of the data in DFS
## Not run:  DD.replicate(data.name)
  DD.list(data.name)

## End(Not run)

# Persist the data to disk
## Not run:  DD.persist(data.name, "file")


# Read data subsets, all subsets in this example
## Not run:  tmp <- DD.get(data.name, NULL)
  length(tmp)
  table(is.na(tmp[[1]]$x))
  DD.close(data.name)

## End(Not run)

# Stop the SupR context
## Not run:  SuprContext(shutdown)


# Double check
```

```
## Not run:  SuprContext()

  DD.open(data.name)
  tmp <- DD.get(data.name, NULL)
  length(tmp)
  table(is.na(tmp[[1]]$x))

  DD.close(data.name)

  SuprContext(shutdown)

## End(Not run)
```

Rython *SupR Thread*

Description

Do SupR Multithreading with thread functions. See `grep("[Tt]hread",ls("package:supr2"),value=TRUE)`.

Usage

```
thread.new(expr, ..., parent.env = parent.frame(), error.handler = NULL,
    name = NULL, proc = FALSE, start = FALSE)
```

Arguments

env	the environment where the expr is to be evaluated.

Details

...

Value

...

Note

...

See Also

Concurrency.

Examples

```
#library(supr2)
#\dontrun{Supr(verbose=TRUE)}

# A Python interactive REPL session within SupR:
## Not run:
  library(supr2)
  py.init()
  py.repl()

  import numpy as np
  import matplotlib.pylab as plt
  x = np.linspace(-np.pi, np.pi, 201)
  plt.plot(x, np.sin(x))
  plt.xlabel('Angle [rad]')
  plt.ylabel('sin(x)')
  plt.axis('tight')
  plt.show()

## End(Not run)

# SupR interface to Python:
## Not run:
####################################

  library(supr2)
  py.init()
  np  <- py.import("numpy")
  plt <- py.import("matplotlib.pylab")
  x   <- np$linspace(-pi, pi, 201L)

  plt$plot(x, np$sin(x))
  plt$xlabel('Angle [rad]')
  plt$ylabel('sin(x)')
  plt$axis('tight')
  plt$show()

## End(Not run)

## Not run:

  library(supr2)
  py.init()
  np  <- py.import("numpy")
  plt <- py.import("matplotlib.pylab")

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    plt.plot   <- plt$plot
    np.sin     <- np$sin
    np.linspace<- np$linspace
    plt.xlabel <- plt$xlabel
    plt.ylabel <- plt$ylabel
```

```
    plt.axis   <- plt$axis
    plt.show   <- plt$show
#-----------------------------------

  x   <- np.linspace(-pi, pi, 201L)
  plt.plot(x, np.sin(x))
  plt.xlabel('Angle [rad]')
  plt.ylabel('sin(x)')
  plt.axis('tight')
  plt.show()

####################################

## End(Not run)
```

SupR *SupR Thread*

Description

SupR: an experimental R package for parallel and distributed computing

Usage

```
SuprContext(what, ...)

ClusterEnv
SharedEnv
```

Arguments

what	...
...	....

Details

SuprContext provides a simple way of working with SupR cluster context. It takes the following arguments:

	Argument	Action
.	driver	shows the driver's connections
.	dfs	shows the DFS_namenode's connections
.	shutdown	stops the cluster context
.	...	...

ClusterEnv is an active binding, returning an object of the SuprShared class. It is used to get and set objects, through the `$` and `$<-` operators, in the cluster environment. See examples.

SharedEnv is an active binding, returning an object of the SuprShared class. It is used to get and set objects through the `$` and `$<-` operators, for threads of three different types, namely, simple thread, local process-based, and (experimental) remote process-based, in a SupR (user) session. See examples.

The access to objects in both ClusterEnv and SuprShared is automatically synchronized. The actual environment referenced by SuprShared is the object SuprShared in .GlobalEnv, whereas the actual environment referenced by ClusterShared is an enviroment object in the Driver session in an alive SuprContext.

Value

...

Note

...

See Also

Monitor, Concurrency.

Examples

```
## start
## Not run: SuprContext()

## Not run:
  SharedEnv$a <- 1:10
  SharedEnv$a

  ClusterEnv$x <- 11:20
  ClusterEnv$x

## End(Not run)

## A local process-based thread gets and sets shared objects.
## Not run:
 SharedEnv$y <- 1:10
 proc <- thread.new({
      SharedEnv$x <- 1:100
      y <- SharedEnv$y
      print(y)
   }, proc=TRUE, start=TRUE)

 thread.join(proc)

 SharedEnv$x
```

```
## End(Not run)

##
```

Thread *SupR Thread*

Description

Do SupR Multithreading with thread functions. See `grep("[Tt]hread",ls("package:supr2"),value=TRUE)`.

Usage

```
thread.current()
thread.env(thread)
thread.expr(thread)
thread.interrupt(thread)
thread.join(thread)
thread.name(name)
thread.new(expr, ..., parent.env = parent.frame(), error.handler = NULL,
    name = NULL, proc = FALSE, start = FALSE)
thread.openXTerm(...)
thread.stackTrace(thread)
thread.start(thread)
```

Arguments

`env`	the environment where the `expr` is to be evaluated.
`expr`	an object to be evaluated by the thread. See `Details`.
`parent.env`	the parent environment of the thread `environment`.
`thread`	a `Thread` or `character` object. The character argument is interpreted as the name of a thread object.
`monitor`	an object whose monitor lock will be acquired for thread synchronization.
`name`	a `character` object as a thread name.
`error.handler`	a function to be called when errors occur. See `Details`.
`proc`	a `logical` object specifying the thread type. See `Details`.
`start`	a `logical` object specifying that thread should start to run right after it is created.
`...`	named objects. For `thread.new`, they are passed in to the thread environment. For `thread.openXTerm`, they are used as arguments for system calls to create X terminals (by threads).

Details

`thread.current` returns the thread object of the calling thread.

`thread.env(thread)` returns the evironment of thread.

...

Value

...

Note

...

See Also

Monitor, Concurrency.

Examples

```
library(supr2)

#\dontrun{Supr(verbose=TRUE)}

thread.check()

## create a new thread to evaluate {print(thread.current()); sin(1:10)}
## Not run:
s <- thread.new({
 print(thread.current())
 sin(1:10)
       })

## End(Not run)

## display thread objects
## Not run:
showThreads()

## End(Not run)

## start the thread and obtain its result
## Not run: thread.start(s)
## Not run:
thread.start(s)

env <- thread.join(s)
print(env)

## End(Not run)
```

ThreadSynchronization *SupR Thread*

Description

Functions for thread synchronization. See `grep("(sync|monitor)",ls("package:supr2"),value=TRUE)`.

Usage

```
monitor.info(...)
sync.eval(monitor, expr, env = parent.frame())
sync.notify(monitor, all = FALSE, env = parent.frame())
sync.wait(monitor, timeout = 60,  env = parent.frame())
```

Arguments

`env`	the environment where the `expr` is to be evaluated.
`expr`	an object to be evaluated by the thread. See `Details`.
`monitor`	an object whose monitor lock will be acquired for thread synchronization. See `Details`.
`all`	logical: if `TRUE`, wake up all the threads waiting on the monitor; otherwise, wake up one of the threads, if any, waiting on the monitor.
`timeout`	numerical: the timeout (in seconds) to be used for synchronized wait of the calling thread.
`...`	a list of objects.

Details

A `monitor` object assembles a pair of `pthread_mutex_t` and `pthread_cond_t` objects for SupR thread synchronization. `monitor.info`, when called with no arguments, shows the list of monitor objects. When called with arguments, it shows the monitor attached to each object in the list of arguments.

The *sync.eval* function does synchronized evaluatoin of its `expr` argument on `monitor`. That is, it steps through four steps: 1. lock (the mutex of) `monitor`, 2. evaluate `expr` in `env`, 3. unlock `monitor`, and 4. return the evaluated value of `expr`.

The *sync.wait* function pauses the execution of the calling thread, so-called sleeps on `monitor`, until woken up by other threads with `notify` calls on `monitor`, or the specified `timeout` has elapsed if `timeout` > 0.

The *sync.notify* function, when called with `monitor` = `FALSE`, wakes up at most one of threads sleeping on `monitor`. When call with `monitor` = `TRUE`, it wakes up all threads sleeping on `monitor`.

Value

...

Note

In case it is necessary, monitors can be unlocked by `.Call("Sync_unlock",monitor)`.

See Also

Monitor, Concurrency.

Examples

```
## FIXME: tools::checkDocStyle(package = "supr2")
## FIXME: tools:::.load_package_quietly("supr2", NULL)
## Not run:
  # a <- tools:::.load_package_quietly
  a <- function (package = "supr2", lib.loc = NULL) {
    if (package != "base")
        .try_quietly({
            pos <- match(paste0("package:", package), search())
            if (!is.na(pos)) {
                detach(pos = pos, unload = package
                  "tools"))
            }
    print("OK 1")
            library(package, lib.loc = lib.loc, character.only = TRUE,
                verbose = TRUE)
        })
  }
  environment(a) <- environment(tools:::.load_package_quietly)

## End(Not run)

## Not run:  # options(editor="vi")

  search()

  detach("package:supr2", unload=TRUE)
  library.dynam.unload("libsupr", paste0(.libPaths()[1],"/supr2"))

  library("supr2", lib.loc = NULL, character.only = TRUE, verbose = TRUE)

## End(Not run)


## library(supr2)

## Make it to work for R CMD build, check, and install
Supr(check = TRUE)

## print(ls(all=TRUE))
```

```
## Not run:
  Supr(verbose=TRUE)
  sync.eval(print(thread.current()))
  monitor.info()
  monitor.info(.GlobalEnv)

## End(Not run)

## Not run:
  #SharedEnv$my.monitor <- "any object that can have attributes"
  SharedEnv$my.monitor <- "any object that can have attributes"

  expr <- quote({
          print(ls(environment(), all=TRUE))
          sync.eval(SharedEnv$my.monitor, {
  message("Wait at ", date())
                  rc <- sync.wait(SharedEnv$my.monitor, 120 + x * 10)
  message("Wokenup at ", date(), " rc = ", rc)
            })

  if(x > 0){
            warning("testing 1")
            warning("testing 2")
  }
  if(x > 1) stop("99")
  x
  })

  th <- as.list(1:3)

  for(i in 1:3)
    th[[i]] <- thread.new(expr, x = i-1L, proc=TRUE, error.handler = print)

  for(i in 1:3) thread.start(th[[i]])

  monitor.info()
  monitor.info(SharedEnv$my.monitor)

  for(i in 1:3) {
    Sys.sleep(2)
    message("notify at ", date())
    sync.eval(SharedEnv$my.monitor, sync.notify(SharedEnv$my.monitor))
    #sync.notify(SharedEnv$my.monitor)
    # more testing is needed ...
  }

  for(i in 1:3) {
    out <- thread.join(th[[i]])
    print(out$.value)
  }

## End(Not run)
```

```
## Not run:
  shared.obj <- new.env()

  expr <- quote({
          print(ls(environment(), all=TRUE))
          sync.eval(shared.obj, {
  message("Wait at ", date())
                  rc <- sync.wait(shared.obj, 5 + x * 5)
  message("Wokenup at ", date(), " rc = ", rc)
           })
  if(x > 0){
            warning("testing 1")
            warning("testing 2")
  }
  if(x > 1) stop("99")
  x
  })

  th <- as.list(1:3)

  for(i in 1:3)
    th[[i]] <- thread.new(expr, x = i-1L, proc=FALSE, error.handler = print)

  for(i in 1:3) thread.start(th[[i]])

  monitor.info()

  for(i in 1:3) {
    Sys.sleep(2)
    message("notify at ", date())
    sync.eval(sync.notify(shared.obj), monitor = shared.obj)
    #sync.notify(shared.obj)
  }

  for(i in 1:3) {
    out <- thread.join(th[[i]])
    print(out$.value)
  }

## End(Not run)


## create a new thread to evaluate {print(thread.current()); sin(1:10)}
s <- thread.new({print(thread.current()); sin(1:10)})

print(search())
print(environment())
print(ls(all=TRUE))

## display thread objects
showThreads()

## start the thread and obtain its result
```

```
## Not run: thread.start(s)
## env <- thread.join(s)
## ls.str(env, all = TRUE)
```

Index