Package 'supr3'

March 15, 2022

Version 22.02.02

Date 2022-02-02

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 (>= 4.0.1)

R topics documented:

---

1. supr3-package *SupR: an experimental R package for parallel and distributed computing*

Description

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

Details

SupR consists of three main components:

1. *Cluster*: a cluster that is based on a collection of stand-alone programs, namely, `driver`, `master`, `worker`, and `taskrunner`. These programs and those for DDS/DFS (`dfs_name` to run as a namenode, and `dfs_data` to run datanodes), are available in package's bin directory. The *context* function provides as a main user interface to start and stop components of a cluster.

   The package provides a set of functions for implementing SupR's method for cluster-wide iterative statistical computing on distributed data, including those known as map-and-reduce or map-and-combine.
2. *SDFS*: a distributed file system, or more generally, a distributed object system. The user interface functions to access DFS have their names starting with *DD*.
3. *Thread*: a set of functions for multithreading, reflecting a collection of experimental efforts to use multithreading in R without any change of the R system code.

Author(s)

Chuanhai Liu

Maintainer: Chuanhai Liu

References

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

See Also

`cluster` `dfs` `thread`

Examples

```
## Not run:

  library(supr3)
  context()
  typeof(context())
  unlist(context()$ths)

  # Turn on verbose mode, 0 <= verbose <= 3L. Large values of verbose
  # increase the verbosity

  supr.options(info=TRUE)  # or supr.options(verbose=1L) [TODO]
  supr.options(debug=TRUE) # or supr.options(verbose=3L) [TODO]

  context(driver = context()$driver)
  context()$driver$connections

  context(shutdown)


## End(Not run)
```

2. SupR *Cluster, DFS, and Thread*

Description

SupR: an experimental R package for parallel and distributed computing.

A cluster is referred to as a system consisting of a `master`, a number of `workers`, and a `driver`. The `master` serves as a central server for workers and drivers. Workers are registered at the `master` and connect to `driver` once a driver is available. A `worker` manages a number of task-runners that are grouped under `executors`. The current implementation of `worker` has a single `executor`. Each task-runner is a R session as a child process of the worker, and works with a `pthread` of the worker to interact with other components of the cluster.

The `driver` provides a user interface to the cluster. Computation tasks are submitted to the cluster as jobs through the driver via the `cluster.eval` function. Details are described in the cluster section.

Usage

```
.SuprEnv
supr.version()
context(...)
supr.options(...)

suprenv(address)
masterenv()
driverenv()
workerenv(which)
nnenv()
dnenv(which)


ping(address, timeout = 10)
netstat(port, args = NULL)

send(msg, to)
recv(timeout = 0)
messages()
```

Arguments

`address`	a character object representing the socket server a SupR component; See *Details*.
`which`	an integer object referring to the worker number registered at the `master` or the DFS datanode number at the DFS `namenode`; See *Details*.
`...`	a list of named arguments reserved for future development.

Details

The `.SuprEnv` variable is an environment object defined in `.GlobalEnv` in every `supr` component R session. It has `.GlobalEnv` as its parent environment, except `taskrunner`. It serves as the parent environment for cluster level evaluations.

The `supr.version` function returns a character object of the `supr` package version.

The `context` function provides a simple way of working with SupR cluster context. It allow the user to start and examine components of a cluster. The arguments of `context` takes the following character (or symbol) values:

Argument	Value	Action
shutdown		command to shutdown cluster by sending a shutdown message to `master`, `driver`, and `dfs_name`.
info	(integer) port_number	start an info socket server
dfs	(character) socket_address	start DFS
	`NULL`	shutdown the dfs server
driver	(integer) port_number	start driver and cluster
	`NULL`	shutdown the driver
master	(character) socket_address	start master and workers
	`NULL`	shutdown the master
ths	(list) socket_addresses	start thread servers
		return the context environment
		session: luster component name
proc		show all supr3 processes
killall		kill all supr3 processes
config	a list of arguments	generate .supr/supr.conf
	nodes=	a character object of hostnames
	force=	TRUE: replace .supr/supr.conf
		FALSE: replace .supr/supr.conf when safe
	port=	lower limit on socket ports to use
	ncpu=	number of cpu corses/taskrunners

The `supr.options` function has three `logical` options `info`, `verbose`, and `debug` and an `integer` option to control the system information details. It offers a `timeout` option that is used in establishing socket connections and reading from sockets. In addition, it has a `port` option, which is used as the lower limit of socket ports when a dynamic socket server port is allowed.

The `suprenv` function creates a socket connection to the server of the cluster component. The target component is specified by its socket server address of the `hostname:port` format. The `driverenv`, `masterenv`, `workerenv`, `nnenv`, and `dnenv` functions are intended to provide a convenient way of accessing cluster components.

The `ping` function sends an echo request to the socket server specified by the address argument.

The `netstat` function uses the OS command `netstat` to provide information on network connections on a given port of interest.

The send function sends an R object msg as a message to the recipient to.

The recv function reads one message, if there are messages available during the time period specified by the timeout (sec.) argument.

The messages function reads all the messages that have been received from the last read.

Value

The suprenv function returns an environment object as an interface to the .SuprEnv object of the component server.

Common objects that are available for all such environments include .SuprThreads, connections, makeActiveBinding, pthreads, shm, and supr.options.

masterenv() returns an environment for accessing master's .SuprEnv.

driverenv() returns an environment for accessing driver's .SuprEnv. In addition to the common objects listed above, this environment has the following driver-specific objects: dfs, master, job, and jobs.

nnenv() returns an environment for accessing DFS namenode's .SuprEnv.

workerenv(which) returns an environment for accessing .SuprEnv of the which-th worker registered at the master. The call workerenv(which=0) returns all the worker addresses.

dnenv(which) returns an environment for accessing .SuprEnv of the which-th datanode registered at the DFS namenode. The call dnenv(which=0) returns all the datenode addresses.

ping(address,timeout) returns a received integer object from the server before timeout is expired, and generates an error message otherwise.

netstat(port) returns a character object as the information on socket connections involved the given port.

recv(timeout) returns the next messages from the last read, if available before the timeout is expired, and NULL otherwise.

messages() returns a list of messages.

Note

The installation of the supr package should create the file named .suprrc, under user's home directory. This is to be run with the bash shell command source to define three system environment variables R_HOME, SUPR_SYS_HOME, and SUPR_USR_HOME. The value of R_HOME is the HOME directory of the R version that was used in the installation. The value of SUPR_SYS_HOME is the HOME directory of the installed supr package. It is used for remote invocations of the supr commands dfs_name, dfs_data, master, worker, and threadserver. The value of SUPR_USR_HOME, with the default $HOME/.supr, is the top-level parent directory of the running supr commands.

Configurations of socket connection ports and distributed file locations are specified in the configuration file: $SUPR_USR_HOME/supr.conf. Here is an example, where a port number n followed immediately with '+' specifies any port in the range from the port n is allowed.

```
Namenode //machine.1:7200?data={
    file://$HOME/.supr/dfs_name/machine.1
```

```
}

Datanode //machine.1:7201?data={
    file://$HOME/.supr/dfs_data/machine.1
    file://$HOME/SupR/dfs_data
}

Datanode //machine.2:7201data={
    file://$HOME/.supr/dfs_data/machine.2
}

Master //machine.1:7202

Driver //machine.1:7203

Worker //machine.1:7204
Worker //machine.2:7204

Infonode //machine.1:7205+

Filetransfer //machine.1:7206+

ThreadServers {
    machine.1:7208+
    machine.2:7208+
}

TrustedHosts {
    machine.1
    machine.2
}
```

Note that multiple files locations can be used for datanodes to store data. This is illustrated with the first datanode in the above example.

The components of supr3 can also be started using machine operation system (OS) commands:

```
dfs_name [-port n] [--verbose] [--debug] [--info] [--help]
         [-info addr] [-notify addr]

dfs_data [-port n] [--verbose] [--debug] [--info] [--help]
         [-info addr] [-notify addr] [-nn addr]

master [-port n] [--verbose] [--debug] [--info] [--help]
       [-info addr] [-notify addr]

worker [-port n] [--verbose] [--debug] [--info] [--help]
       [-info addr] [-notify addr] [-master addr] [-ntrs k]
```

```
driver [-port n] [--verbose] [--debug] [--info] [--help]
       [-info addr] [-notify addr] [-shm name] [-master addr]
       [-dfs addr]


threadserver [-port n] [--verbose] [--debug] [--info] [--help]
       [-info addr] [-notify addr]
```

More information on these commands can be obtained by invoking them with the `--help` argument.

See Also

Cluster, Thread

Examples

```
## start
## Not run:

################################################
##### New functions, subject to more testing #####
################################################

library(supr3)

context(proc)    # show SupR processes on all the nodes found in supr.conf
context(killall) # kill all SupR processes on all the nodes found in supr.conf

# The following function call

context(config=list(nodes="bell-fe[00-06]", force = TRUE, port=2022))

# generates ~/.supr/supr.conf with the given nodes argument and
# the same network domain as that of the host of the R session

# The next example shows how to use nodes obtained via
# an interactive Slurm job, sinteractive -N 10 -n 100 -A standby -t 1:0:0
# Slurm Reference: <https://slurm.schedmd.com/quickstart.html>


context(config= list(nodes = Sys.getenv("SLURM_JOB_NODELIST"),
         force=TRUE, port = 2022, ncpu=10))


# The following function call, subject to feature refinements,

# workerenv(1)$supr.options(xterm = TRUE)
workerenv(1)$xterm(interactive=TRUE)

# starts a remote X Terminal (xterm) for stdout and stderr and
```

```
# an interactive pthread running R REPL (Read-Eval-Print Loop)
# on worker workerenv(1).

# Added Sys.info2

Sys.info2()

# which provides information on the number of cpu cores and the limits
# on the maximum file descriptor number that can be opened by
# current process

################################################
#####          End of New functions          #####
################################################



library(supr3)
supr.options()
supr.options(info = TRUE, verbose = TRUE)

# Show the initial state of the context
context()

# Start a cluster
context(driver = context()$driver)
cluster <- context()$driver
cluster$connections

masterenv()$connections
masterenv()$pthreads
workerenv(0)
workerenv(1)$connections
workerenv(1)$pthreads

nnenv()$connections
nnenv()$pthreads
dnenv(0) # datanode server addresses
dnenv(1)$connections
dnenv(1)$pthreads

# Shutdown the cluster
context(shutdown)

## End(Not run)
```

3. Cluster *SupR Cluster Interface*

Description

Functions to interface with the SupR cluster system.

Usage

```
.GlobalEnv$.SuprEnv$.LastValue
.job.id
.tr.id
JobEnv

clusterenv()
clusterenv()$jobs()
clusterenv()$job(job_id)

cluster.eval(expr, data, envir = list(), error.handler = NULL,
     env = parent.frame(), wait = TRUE, combine = job.combine.default,
     ...)
cluster.combine(x, combine, bykey = FALSE, env = parent.frame())
cluster.get(uri, name, remove = FALSE)
collect(x, remove = FALSE)

nextSubset(env = parent.frame())
info(text, conn = NULL)
# CHANGE IT TO cluster.combine

sync(monitor, expr, language.only = FALSE, env = parent.frame())
wait(monitor, timeout = 0, env = parent.frame())
notify(monitor, all = FALSE, env = parent.frame())

broadcast(..., addr = NULL)
broadcasted(new = FALSE, timeout = 0)
```

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	an environment object.
error.handler	the error handler at the cluster level (not implemented yet)

wait	a logical object.
combine	a function used cluster.eval to combine results from taskrunners or a function that is to be used in cluster.combine to combine.
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.
bykey	a logical object.
x	a named list or environment object.
...	optional arguments.For cluster.eval, they are job-specific variables defined at the cluster level, i.e., at the driver site. For broadcast, they are variables to be sent to taskrunners.

Details

The object JobEnv acts as a UserDatabase linked to a cluster environment object at the driver session. The variables job, task, address, firstSubset, and .__NAMESPACE__. are handled locally. The variable job is an integer object of a job id. When the value of job is -1, JobEnv is referred to as a UserDatabase to access global environment for job variables located at the driver site.

...

Value

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

A job state notification is sent to user as a message (see send, recv, and messages) by the driver. The message is an object of the JobStatus class. Objects of the JobStatus class have three named objects, namely, job, state, and ntrs. The job element is an integer as the job id. The ntrs element is the number of task runners that are still involved in finishing the job. The state element takes one of the three values as follows:

Job State	Notification Value	Comments
TASKS_ASSIGNED	"assigned"	all the subsets have been taken
CANCELLED	"cancelled"	by user or due to errors
FINISHED	"finished"	

With wait = TRUE, cluster.eval returns an environment object, having a S3 class name "Future".

The class "Future" has the following objects to operate on the submitted job.

```
class "Future" {
    cancel(): cancel the submitted job.
    get(timeout = 0): get the result if available within next \code{timeout}
                      seconds.
    is.cancelled(): returns TRUE if the job is cancelled and FALSE otherwise.
    is.done(): return TRUE if the job is finished and FALSE otherwise.
    job.id: an integer object used as the job id.
    .value: the list of returned values from the taskrunners.
    combine: combine the list of taskrunner results, \code{.value}.
    .done: a logical object used internally by the \code{is.done} function.
    envir: the environment object of the \code{env} argument in calling
           \code{cluster()$eval}.
}
```

When a job submitted with `wait = TRUE` is user-interrupted by, for example, pressing Ctr-C on the keyboard, the job will continue as with `wait = FALSE`. The object of class "Future" for accessing this job can be obtained from the current value of `.GlobalEnv$.SuprEnv$.LastValue`.

The three objects named `.job.id`, `.tr.id`, `.tr.name` are defined in the environment, `.GlobalEnv$.SuprEnv`, of taskrunners. The value of `.job.id` object is the integer value of the ID of the job the taskrunner is running. The value of `.tr.id` object is the integer value of the job-specific taskrunner ID assigned to the taskrunner running the job. The value of `.tr.name` object is the character of length one to uniquely identify the taskrunner process. Its value has the form of hostname:pid and equals `paste(Sys.info()["nodename"],Sys.getpid(),sep=":")`.

Note

...

See Also

`dfs`, `thread`

Examples

```
## Not run:

  library(supr3)
  cluster <- clusterenv()
  supr.options(level = 1)

  ls(cluster)
  ls(cluster$.LocalEnv)
  cluster$connections
  cluster$pthreads
  cluster$supr.options()
  cluster$supr.options(level=1)
```

```
## End(Not run)

# An echo example of different kinds
## Not run:

## a plain version
  expr.echo <- quote({
     subsets <- list()
     while(!is.null(x <- nextSubset()))
        subsets <- append(subsets, x)
     subsets
  })
  results <- cluster.eval(expr.echo, as.list(1:100), envir=list())
  sort(unlist(results))
  messages()


## a version making use of the send function
  expr.echo <- quote({
     subsets <- list()
     while(!is.null(x <- nextSubset())){
        send(x, "user")
        subsets <- append(subsets, x)
     }
     subsets
  })
  results <- cluster.eval(expr.echo, as.list(1:100))
  sort(unlist(results))
  messages()


## a version making use of the send and recv functions
  expr.echo <- quote({
     subsets <- list()
     while(!is.null(x <- nextSubset())){
        Sys.sleep(1)
        send(x, "user")
        subsets <- append(subsets, x)
     }
     subsets
  })

  eval({
    results <- cluster.eval(expr.echo, as.list(1:100), wait=FALSE)

    i <- 0
    y <- numeric(0)
    while(!is.null(x <- recv(timeout=60))){
      if(class(x) == "JobStatus"){
        print(x)
        if(x$state == "finished") break;
      } else {
        cat("[", i <- i + 1,"] x:", x, "\n")
```

```
        y[i] <- x
        plot(1:i, y)
      }
    }

    sort(unlist(results$get()))
  })


## End(Not run)


## a version making use of JobEnv
## Not run:
  library(supr3)
  cluster <- clusterenv()
  supr.options(level=1)

  expr.echo <- quote({
     subsets <- list()
     while(!is.null(x <- nextSubset())){
        JobEnv$z <- x
        Sys.sleep(1)
        info(paste("JobEnv$i =",JobEnv$i));
        send(x, "user")
        subsets <- append(subsets, x)
     }
     subsets
  })

  eval({
    results <- cluster.eval(expr.echo, as.list(1:100), wait=FALSE)

    JobEnv$job  <- results$job.id
    JobEnv$task <- -1
    i <- 0
    y <- numeric(0)
    z <- numeric(0)
    while(!is.null(x <- recv(timeout=60))){
      if(class(x) == "JobStatus"){
        print(x)
        if(x$state == "finished") break;
      } else {
        cat("[", i <- i + 1,"] x:", x, "\n")
        JobEnv$i <- i
        y[i] <- x
        z[i] <- JobEnv$z
        plot(1:i, y)
        lines(1:i, z)
      }
    }

    sort(unlist(results$get()))
```

```
  })

  cluster$jobs()
  context(shutdown)

## End(Not run)


## Not run:
# The use of the parallel random number generator in the rlecuyer package
  library(supr3)
  cluster <- clusterenv()
  supr.options(level=1)

  expr.rgn <- quote({ # FIXME?
          library(rlecuyer)
  retval <- list()
          while(!is.null(x <- nextSubset())) {
             info(paste("x:", x, "pid:", Sys.getpid()))
             .lec.SetPackageSeed()
             .lec.StreamNames <- paste("lec.stream", 1:ntasks, sep=".")
             .lec.CreateStream(.lec.StreamNames)
             kind.old <- .lec.CurrentStream(.lec.StreamNames[.tr.id])

             val <- list(ntasks=ntasks, x=x,tr.id=.tr.id,
    u.1 = runif(1), u.2 = runif(1))

             .lec.CurrentStreamEnd(kind.old)
             .lec.DeleteStream(.lec.StreamNames)
             retval <- rbind(retval, val)
  }
          info(paste("x:", x, "pid:", Sys.getpid()))
  retval
       })

  results <- cluster.eval(expr.rgn, as.list(1:100L),
                          envir=list2env(list(ntasks=100L)), wait=F)


## End(Not run)

# A cluster-level combine example
## Not run:

  library(supr3)
  cluster <- clusterenv()
  supr.options(level=1)

  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))

    results
    unlist(results)

    cluster$jobs()
    context(shutdown)

  ## End(Not run)

  # A cluster-level combine_by_key example

  ## Not run:

    library(supr3)
    cluster <- clusterenv()
    supr.options(level=1)

    data <- 1:100
    data <- lapply(as.list(data), function(x) {
                      names(x) <- LETTERS[(x-1) %% 26 + 1]
                      x
                    })

    expr.cbk <- quote({
      Sys.sleep(1)
      tr.combine <- function(x, y){
         keys <- unique(c(names(x), names(y)))
         z <- numeric(length(keys))
         names(z) <- keys
         for(key in keys)
   z[key] <- sum(c(if(is.null(x)) x else x[names(x)==key],
   if(is.null(y)) y else y[names(y)==key]))
         z
      }
      y <- NULL
      while(!is.null(x <- nextSubset()))
          y <- tr.combine(y, x)

      data <- as.environment(as.list(y))
      send(ls.str(data), "user")
      my.combine <- function(x, y) x+y
      cluster.combine(data, my.combine, bykey = TRUE)
    })

    results <- cluster.eval(expr.cbk, data=data)

    messages()
```

```
  r <- collect(results, TRUE)
  r <- unlist(r)

  # Checking
  s <- sapply(LETTERS, function(x) {
             sum(unlist(lapply(data, function(y) y[names(y)==x])))
       })
  s <- s[sort(names(s))]

  # Checking: is r[names(s)] - s a vector of 26 zeros?
  print(r[names(s)] - s)



## End(Not run)



# A simple EM example using the standard synchronization method.
# For the DD object EM_data, see the DD example
## Not run:
  options(verbose=TRUE)
  options(debug=TRUE)

#  options(info=TRUE) # info default is TRUE
  library(supr3) # .Call("Pthread_sigmask", NULL)
  context(driver = context()$driver)
  cluster <- context()$driver
  cluster$connections

  expr.em <- quote({ subsets <- list()
    while(!is.null(sub <- nextSubset())) { x <- sub$x[,1]
      subsets <- append(subsets, list(x)); Sys.sleep(0.1)
# if(runif(1)<0.01) stop("testing")
    }
    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(length(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 = 10; this.job <- paste0("JOB-", .job_id)
    while(n < max.n) { n <- n  + 1
      SS <- E.step(mu) #my.result <- cluster.combine(SS, my.combine)
```

```
      my.result <- cluster.combine(SS, my.combine)
      if (is.null(my.result)) {
        mu <- sync(std_EM, {timeout <- wait(std_EM, timeout=timeout())
  if(timeout < 0) notify(std_EM, all=TRUE)
  JobEnv$mu })
#if(runif(1)<0.01) stop("testing") else if(runif(1)<0.01) q("no")
      } else {
        mu <- M.step(my.result) #broadcast(mu=mu, test="OKAY?")
JobEnv$mu <- mu
        sync(std_EM, { timeout <- wait(std_EM, timeout=timeout())
   if(timeout < 0) notify(std_EM, all=TRUE)
  s <- if(timeout < 0) "*" else " "
        #send(paste(s, "job.id =", .job_id, "iter =", n, "mu =", mu), "user")
        info(paste(s, "job.id =", .job_id, "tr.id =", .tr_id,  "iter =", n, "mu =", mu))
  })
     }
    }
    list(mu=mu, Sx=SS$sx, n = n, n.mis=sum(mis), n.subsets=n.subsets)
 })
 print(proc.time())
 result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"))
 print(proc.time())

for(i in 1:5000){print(i); result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"), wait=TRUE) }


for(i in 1:10){print(i); result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"), wait=FALSE) }

 # CHECKING job_id > 60?
 options(info=TRUE)
 library(supr3)
 context(driver = context()$driver)
 cluster <- context()$driver
 cluster$connections


 ## Use broadcast for sync
 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(length(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 = 50; this.job <- paste0("JOB-", .job_id)
    while(n < max.n) { n <- n  + 1
      SS <- E.step(mu) #my.result <- cluster.combine(SS, my.combine)
      my.result <- cluster.combine(SS, my.combine)
      if (!is.null(my.result)) {
        mu <- M.step(my.result)
broadcast(mu=mu, iter = n)
        info(paste("pid =", Sys.getpid(), "job.id =", .job_id,
   "tr.id =", .tr_id, "iter =", n, "mu =", mu))
# info(as.character(rc))
      }
#      if(runif(1)<0.01) stop("testing")
      mu <- broadcasted(new = TRUE)$mu
    }
    list(mu=mu, Sx=SS$sx, n = n, n.mis=sum(mis), n.subsets=n.subsets)
  })

  print(proc.time())
  result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"))
  print(proc.time())


 i <- 0; while(TRUE){i <- i+1; print(i); result <- cluster.eval(expr.em, data=structure("EM_data", class="DD")) }


  library(supr3)
  cluster <- clusterenv()
  supr.options(level=1)

  ## Use broadcast for sync
  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(length(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; this.job <- paste0("JOB-", .job_id)
    while(n < max.n) { n <- n  + 1
      SS <- E.step(mu) #my.result <- cluster.combine(SS, my.combine)
```

```
      my.result <- cluster.combine(SS, my.combine)
      if (!is.null(my.result)) {
        mu <- M.step(my.result)
broadcast(mu=mu, iter = n)
        send(list(mu=mu, n=n), "user")
      }
      mu <- broadcasted(new = TRUE)$mu
    }
    list(mu=mu, Sx=SS$sx, n = n, n.mis=sum(mis), n.subsets=n.subsets)
  })

  messages()
  res <- eval({
    result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"),
   wait = FALSE, envir=list(max.n = 50), my.par=1:10)
    i <- 0; mu <- n <- numeric(0)
    while(!is.null(x <- recv(timeout=5))){
      if(class(x) == "JobStatus"){
        print(x)
        if(x$state == "finished") break;
      } else {
        i <- i+1
        mu[i] <- x$mu
        n[i] <- x$n
        s <- ceiling(length(n)/2):length(n)
        plot(n[s], mu[s])
      }
    }
    result$get()
  })

  messages()



## End(Not run)

## An asynchronous distributed EM
## Not run:
  library(supr3)
  cluster <- clusterenv()
  supr.options(level=1)

  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(length(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) }

    n <- 0; this.job <- paste0("JOB-", .job_id)
    while(n < max.n) { n <- n  + 1
      SS <- E.step(mu)
      send(list(SS=SS, tr = .tr_id), "user")
      mu <- broadcasted(new = TRUE)$mu
      info(paste("mu:", mu))
    }
    list(mu=mu, Sx=SS$sx, n = n, n.mis=sum(mis), n.subsets=n.subsets)
  })

  messages()
  res <- eval({

    M.step <- function(SS){
    sum(unlist(lapply(SS, "[[", "sx")))/
    sum(unlist(lapply(SS, "[[", "n")))
    }
    result <- cluster.eval(expr.em, data=structure("EM_data", class="DD"),
   wait = FALSE, envir=list(max.n = 50), my.par=1:10)

    JobEnv$job  <- result$job.id; JobEnv$task <- -1
    ##ntrs <- 0; SS <- list(); R <- logical(0)

    n <- 0; mu.seq <- n.seq <- numeric(0)
    while(!is.null(x <- recv(timeout=5))){
      if(class(x) == "JobStatus"){
print(x)
        if(x$state == "finished") break;
        if(x$state == "assigned") {
       ntrs <- x$ntrs
SS <- lapply(as.list(1:ntrs), function(x) list(sx=0, n=0))
R <- logical(ntrs)
count <- logical(ntrs)
}
      } else {
        k <- x$tr+1
        SS[[k]] <- x$SS
R[k] <- TRUE
if(sum(R) >= 0.75*ntrs || runif(1) < 0.01){
  cat("sum(R):", sum(R), "\n")
  mu <- M.step(SS)
  n <- n + 1
  broadcast(mu=mu, iter = n)
  R[] <- FALSE
         mu.seq[n] <- mu
         n.seq[n] <- n
```

```
          s <- ceiling(length(n.seq)/2):length(n.seq)
          plot(n.seq[s], mu.seq[s])
}
      }
    }
    if(is.null(x)) stop("unexpected NULL msg ...")
    # result$get()
  })

  # result$cancel()


## End(Not run)
```

4. DDS or DFS *SupR Distributed Data System*

Description

These functions represent an interface for accessing the SupR distributed data system (DDS). They include

the common set of functions, namely, `assign` or `` `$<-` ``, `exists`, `get` or `` `$` ``, `ls`, and `rm`, as an interface for `environments` and `userDefinedDatabase` objects.

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, create = TRUE)
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)

DDEnv(name=".", addr = context()$dfs)
open.DDEnv(ddenv, name, save = TRUE)
close.DDEnv(ddenv, name, save = TRUE)
is.DDEnv(x)
print.DDEnv(x, ...)
replicate.DDEnv(x)
mv(src, dest)

`[.DDEnv`(x, i, j, combine, ..., env = list())
```

Arguments

`create`	a logical object.
`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 evaluations of arguments.
`src`, `dest`	DD names.
`level`	one of the values given in its default.

subsets	a list of objects representing subsets.
subset.names	a character vector specifying the subset names.
...	optional arguments.

Details

The DD() function makes an DD object, regardless the existence of the named DD.

The DD.exists() function tests for the existence of the distributed data.

The DD.get() function gets the named data subsets of the DD name from DFS.

The DD.list() function returns simple information on the DD name or a file directory at DFS.

The DD.mv() function rename src to dest.

The DD.open() function opens the data data named name if it exists. Otherwise, it creates and opens the data if create = TRUE, and generates an error if create = FALSE.

The DD.options() invokes the options() function at DFS*.

The DD.persist() changes the storage type of the named DD.

The DD.put() function adds data subsets, a list of objects, with subset names subset.names to the distributed data named name.

The DD.replicate() increases the (minimum) number of subset replications by one.

The DD.rm() removes the named DD.

Value

The DD() function returns an DD object, regardless the existence of the named DD.

The DD.close() function returns an integer as the return code* The DD.exists() function returns TRUE if the named DD exists, and FALSE otherwise. The DD.get() function returns a list of objects. The DD.list() function returns a data frame object of subset name, size, timestamp, locations, and number of replications. The DD.mv() function returns an integer as the return code* The DD.open() function returns a DD object.

The DD.options() function returns the value of the options() function call at DFS*.

The DD.persist() function returns an integer as the return code*

The DD.put() function returns the number of subsets.

The DD.replicate() function returns the number of replications of the updated DD object.

The DD.rm() function returns an integer as the return code*

The DDEnv function returns a DDEnv object as an alternative representation of a DD object, in term of an userDefinedDatabase and environment object. The other functions following DDEnv on above usage list of functions are intended to provide operations to act on DD objects in the same way as on environment objects, as demonstrated below in the examples.

The `[.DDEnv` function provides a simple way to operate on DDEnv objects as distributed data.frame or data.table. The general form of DDEnv syntax is DDEnv[i,j,combine,...,env = list()]. The i argument is an expression. When evaluated remotely, its value is either a logical or integer object, which is to be use as to select rows of DDEnv. The j argument is an expression to be evaluated remotely. The logical, integer, and character values are used to select columns according

to column variable positions and names. When its value is a list, a new `data.frame` or `data.table` type of object is created with the list of objects, when its components are objects of `logical`, `integer`, `numeric`, and `character` and have the same length.

Note

...

See Also

`Cluster`, `Supr`, `Thread`.

Examples

```
# Create a distributed data, named "EM_data", consisting of
# simulated subsets, and upload it into DFS
## Not run:
  options(info=TRUE)
  library(supr3)

  context(dfs=context()$dfs)

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

    DD.close(data.name)

    supr.options(verbose=TRUE)

    x <- ddenv("EM_data")
    x[][]
  }


  ## distribute subsets of R objects saved in files
  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)

## distribute R objects via R serialization
## Not run:
  options(info=TRUE)
  library(supr3)
  context(dfs=context()$dfs)

  dn <- dnenv(1)
  dn$supr.options(level=1)
  dn <- dnenv(2)
  dn$supr.options(level=1)

  dfs <- context()$dfs
  nn <- nnenv()
  nn$supr.options(level=1)



  data.name = "data_frame"

  if(DD.exists(data.name)) DD.rm(data.name, TRUE)
  DD.open(data.name)
  N <- 10L  # number of subsets
  K <- 1000L # subset size
  P <- 10L # number of variables
  var.names <- paste("X", 1:P, sep=".")
  for(i in 1:N){
    X <- matrix(runif(K*P), nrow=K, dimnames=list(NULL, var.names))
    X <- as.data.frame(X)
    DD.put(data.name, list(X), paste0("part_", i))
    # added suffix ".sro" internally
  }

   DD.list(data.name)
   DD.persist(data.name)
```

```
   DD.get(data.name, "part_2.sro")

   A <- dfs$open(data.name)
   A[1:2, 2:4]


## 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")
  DD.close(data.name)

## End(Not run)

# 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:  context(shutdown)


# Double check
## Not run:  context()

  Supr(verbose = TRUE)
  data.name = "EM_data"
  DD.open(data.name)
  tmp <- DD.get(data.name, NULL)
  length(tmp)
  table(is.na(tmp[[1]]$x))

  DD.close(data.name)

  context(shutdown)

## End(Not run)

## Not run:
  # library(supr3)
  # context(dfs=context()$dfs)
```

```
  options(debug=TRUE)

  options(info=TRUE)
  library(supr3)
  context(dfs = context()$dfs)
  dfs <- context()$dfs
  dfs <- ddenv()

  {
    a <- dfsenv()
    a$connections
    a$pthreads
  }


  {
  em.data <-  dfs$open("EMData")
  }

  context(dfs = NULL)


   em.data <- open(DDEnv("EMData"))

   assign("x", 1:10, envir = em.data)
   assign("y", 1:100, envir = em.data)
   ls(em.data)
   get("x", envir=em.data)
   get("y", envir=em.data)
   get("z", envir=em.data)
   em.data$x
   em.data$y
   em.data$z
   em.data$z <- 1:20
   em.data$z

   exists("EMData", envir=em.data) # check subsets... or no?

   rm(EMData, envir = em.data) # check subsets... or no?

   # mv(em.data, as.character(where))
   persist(em.data, level)
   replicate(em.data)


 if(FALSE){

  ################################################################
  ## some limited experiments on `[.DDEnv`                      ##
  ## a tada.table framework; See the data.table R package     ##
  ################################################################

  options(info=TRUE)
```

```
library(supr3)
context(dfs = context()$dfs)
#supr.options(level=1)

dn <- dnenv(1)
dn$supr.options(level=1)
dn <- dnenv(2)
dn$supr.options(level=1)

dfs <- context()$dfs
nn <- nnenv()
nn$supr.options(level=1)

#  em.data <-  dfs$open("EMData")
em.data <-  dfs$open("EM_data")

b <- (a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), , foo= sin])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), test,
   test= function(x) {
     v <- x$value
        x <- x$x
if (is.null(x)) {
return(v)
}
else if (class(x) == "DD.frame") { # dn
n <- length(x)
append(v, as.list(n))
} else { # nn
append(v, x)
}
  }, foo= sin][,,dim])


(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), test, test= function(x) {
   c(x$value, runif(10))
  }, foo= sin])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), dim , foo= sin])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), , foo= sin][, 2:3, dim, j2=4])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), , foo= sin][, 2:3, , j2=4][,,dim()])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), , foo= sin][, 2:3, , j2=4][, , ][,,dim()])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), function(x) {
 if(runif(1)<0.1) stop("testing")
   c(x$value, search())
  }, foo= sin])

(a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), function(x){if(runif(1) < 0.01) stop("TESTING"); 100} ,
```

```
 (a <- em.data[!is.na(x), list(x=x, y=x^2, z = foo(x), key = x > 0), dim, foo= sin][, 1:3, , j2=4][, 1:2, group=funct
 )

 (a <- em.data[, list(x=x, y=x^2, z = foo(x), key = x > 0), .n, foo= sin][, 3, , j2=4][, , group=function(){} ][1,2,3
 )

   a
}

  # close(em.data)
  {
    dfs$close(attr(em.data, "name"))
    em.data$close()
  }

  context(shutdown)

## End(Not run)
```

5. Thread *SupR: Thread*

Description

SupR Multithreading with thread functions, implemented using both (Linux) processes and pthreads.
To see the exported functions, run the code `grep("[Tt]hread",ls("package:supr3"),value=TRUE)`.

Usage

```
ThreadServer.cancel(host, port, address = paste(host, port, sep = ":"))
ThreadServer.connect(host, port, address = paste(host, port, sep = ":"))
connect(host, port, address = paste(host, port, sep = ":"))
ThreadServer.start(port)

Thread.interrupt(thread)
interrupt(thread)
Thread.join(thread, useProxy = TRUE)
join(thread, useProxy = TRUE)
Thread.new(expr, ..., env = parent.frame(), error.handler = NULL,
    server = local.thread.server, start = FALSE)
ThreadServer.connect(...)$new(expr, ..., env = parent.frame(), error.handler = NULL,
    server = local.thread.server, start = FALSE)
start(thread)
state(thread)

Sync.eval(monitor, expr, language.only = FALSE, env = parent.frame())
Sync.notify(monitor, all = FALSE, env = parent.frame())
Sync.wait(monitor, timeout = 0, env = parent.frame())
sync(monitor, expr, language.only = FALSE, env = parent.frame())
notify(monitor, all = FALSE, env = parent.frame())
wait(monitor, timeout = 0, env = parent.frame())
```

Arguments

`env`	the parent of the environment where the `expr` is to be evaluated.
`error.handler`	a function to be called when errors occur. See `Details`.
`expr`	an object to be evaluated by the thread. See `Details`.
`host`	a `character` representing the hostname of the thread server.
`port`	an `integer` representing the socket port of the thread server.
`monitor`	an expression of the form `connection$name` specifying a monitor named `name` at the ThreadServer `connection`.
`thread`	a `Thread` object created by `Thread.new`.
`start`	a `logical` object specifying that thread should start to run right after it was created.
`...`	named objects. For `Thread.new`, they are passed on to the thread environment.

Details

ThreadServer.start is used to start a thread server in the current R session.

ThreadServer.connect or connect is used to connect to the thread server specified by the arguments hostname host and socket port number port.

ThreadServer.cancel cancels the thread server specified by the arguments hostname host and port number port.

Thread.new creates a new thread in the thread server session. Non-name expr objects are to be evaluated by the thread. Name expr objects are looked up in the current scope and their bindings are to treated as non-name expr objects.

Thread.start(thread) starts the thread identified by the thread object.

Thread.join(thread) or join(thread) obtains the result of the thread. This should be always called because the system resources allocated for the thread can only be released via this function call, which invokes the C library function waitpid etc.

Thread.interrupt(thread) or interrupt(thread) terminates the running of the thread.

Thread.state(thread) or state(thread) gets the last reported running state of the thread.

Sync.eval obtains the monitor and evaluates the expression expr while holding the monitor.

Sync.wait or wait releases the monitor and waits to be notified or for a time period specified by the argument timeout in seconds. The special value timeout = 0 means that no time limit is provided. timeout can take an expression to be evaluated by the synchronization manager such as thread servers and driver in cluster evaluation. In this case, the value of the expression must be numeric and is used to determine the required time period (in seconds).

Sync.notify or notify wakes up threads waiting on the same monitor one at a time if all = FALSE and all with a single call with all = TRUE.

Value

ThreadServer.start returns a ThreadServer object, containing its hostname and port number for service connections.

ThreadServer.connect or connect returns a ThreadServer connection object. This object is to be used for the service of SupR-threading. It servers also as an environment object for accessing shared objects at the thread server session.

ThreadServer.cancel returns 0 if successful and -1 otherwise.

Thread.new return a thread identification number with the thread server attached as an attribute named server.

Thread.start(thread) returns an integer object 0 if successful and -1 otherwise.

Thread.join(thread) or join(thread) returns the value of thread evaluation along with the outputs to stout and stderr if the task was performed normally. When errors occur, the returned object has no value. In such a case, the error message, if available, is attached as an attribute named error.

Thread.interrupt(thread) or interrupt(thread) returns 0 if successful and -1 otherwise.

Thread.state(thread) or state(thread) returns an integer value as the state of the thread as follows: 0 — created, 1 — started, 2 — evaluated, 3 — finished.

`Sync.eval` returns the value of the expression `expr`.

`Sync.wait` or `wait` returns 0 if successful and -1 otherwise.

`Sync.notify` or `notify` returns 0 if successful and -1 otherwise.

Note

The threading model here follows that of the JAVA language. Each thread is implemented with a Linux process. The process ID is used as the thread ID.

The shared objects at thread servers can be accessed via functions accessing objects in environments. A standard set of such functions consists of `ls`, `$`, `$<-`, `exists`, and `rm`.

See Also

`SupR` for some utility functions; `Distributed File System` for an alternative way of storing and accessing on SupR clusters; and `Cluster` for cluster such as map-and-reduce type of computing.

Examples

```
### NEW ###
library(supr3)

# Generate supr.conf based on SLURM_JOB_NODELIST
# where the syntax of (the value of) SLURM_JOB_NODELIST is
# ...
# context(config= list(nodes = Sys.getenv("SLURM_JOB_NODELIST"),
#    force=TRUE, port = 2022), ncpu=5)

## Not run:
supr.options(info=TRUE, xterm = TRUE)
context(killall)

# Start thread servers

context(ths=context()$ths)
ths <- context()$ths
ls(ths[[1]], all = TRUE)
ls(ths[[1]]$.LocalEnv, all = TRUE)
ths[[1]]$new

# Local info on thread servers
context(ths=list())

# Stop thread servers
context(ths=NULL) # or context(shutdown)



## End(Not run)
```

```
### END_NEW ###

## Load the supr3 package in a R session:


library(supr3)

## Create a thread server in this the current session with a port number in an
## appropriate range.  Keep a record of this server address, consisting of the
## hostname and socket port number for connections from this and other R
## sessions on the same machine or different machines. At most one thread
## server can be running in a single R session. However, multiple thread
## servers can run simultaneously on different R sessions and on different
## machines.

## Not run:
 server <- ThreadServer.start(port=2021L)
 print(server)

context(ths=context()$ths)
ths <- context()$ths


## End(Not run)

## Connect thread servers (only one server in the following example) from
## the same or different R session:

## Not run:
servers <- attr(server, "address")

connections <- as.list(1:length(servers))
for(i in 1:length(servers))
    connections[[i]] <- connect(address = servers[i])


connections <- append(connections, ths)


## End(Not run)

## For a simple example, take one from these connections at random:

## Not run:
conn <- connections[[sample(1:length(connections), size=1)]]
print(conn)


## End(Not run)

## Use the above connection conn as a shared network environment for R objects:
```

```
## Not run:
ls(conn)
conn$x <- 1:10
ls(conn)
conn$x
exists("x", envir = conn)
exists("y", envir = conn)
rm("x", envir = conn)
objects(conn)


## End(Not run)

## Create a new thread to evaluate:
## Not run:
s <- conn$new({
 print(Sys.info()["nodename"])
 print(Sys.getpid())
 sin(1:10)
       })

start(s)
state(s)
res<- join(s)
print(res)


## End(Not run)

## or, equivalently,
## Not run:
expr <- quote({
 print(Sys.info()["nodename"])
 print(Sys.getpid())
 sin(1:10)
       })

s <- conn$new(expr)

start(s)
state(s)
res<- join(s)
print(res)


## End(Not run)


## Start the thread:
## Not run:
start(s)
```

```
## End(Not run)

## Check the running state of the thread:
## Not run:
state(s)


## End(Not run)

## Get the result and automatically release the resources:
## Not run:
res <- join(s)


## End(Not run)


## A simple EM example to demonstrate the use of SupR thread synchronization.
## Create an artificial incomplete data from the standard Gaussian
## distribution with a fractional number of values missing at random:
## Not run:
  n <- 1000; mis.frac <- 0.25
  X <- rnorm(n)
  X[runif(n) < mis.frac] <- NA


## End(Not run)

## Start default thread servers with addresses provided in .supr/supr.conf
## Not run:
   library(supr3)
   supr.options(info=TRUE)
   #or
   supr.options(info=TRUE, xterm=TRUE)

   context(killall)

   context(ths=context()$ths)

   connections <- context()$ths


## End(Not run)


## Start m threads (at random) by the thread servers and choose a single
## server for shared objects
## Not run:
m <- 10
shared <- connections[[sample(1:length(connections), size=1)]]
shared.addr <- attr(shared, "address")
# or
shared.addr <- shared$address
```

```
## End(Not run)

# Create an object named as 'SS' at the thread server 'shared' for storing
# the expected sufficient statistics, sum(X) and length(X):
## Not run:
shared$SS <- c(0,0)


## End(Not run)

## Partition the data into m subsets:
## Not run:
  m <- 10 # the number of threads to be used
  dim(X) <- c(m, n/m)


## End(Not run)

# Create m threads:
## Not run:
  threads <- as.list(1:m)
  for(i in 1:length(threads)){
     conn <- connections[[sample(1:length(connections), size=1)]]
     expr <- quote({
 shared <- connect(address = shared.addr)
         info("Start EM ...")
         which.mis <- which(is.na(x))
         for(it in 1:50){
             #E-step
             x[which.mis] <- mu
             my.SS <- c(sum(x), length(x))
             sync(shared$EM, {            #### a 'Sync.eval' example
                   SS <- shared$SS + my.SS
                   cat("[",it, "] SS:",SS, "n:", n, "\n")
                   if(SS[2] < n) {
                     shared$SS <- SS
                     wait(shared$EM)              #### a 'wait' example
                     mu <- shared$mu
                     cat("[",it, "] mu:",mu,"\n")
                   } else {
                     #M-step
                     mu <- SS[1]/n
                     info(paste("[",it, "] pid:",Sys.getpid(),
                        "mu:",mu))                #### send a text message
                     shared$mu <- mu
                     shared$SS <- c(0,0)
                     notify(shared$EM, all=TRUE)  #### a 'notify' example
                   }
                 })
         }
         info(paste("\033[0;31m[",it, "] pid:",Sys.getpid(),"\033[0m"))
```

```
        shared$mu
    })
    threads[[i]] <- conn$new(expr, x=X[i,],n=n,mu=0,shared.addr=shared.addr)
  }


## End(Not run)

## Start the m threads
## Not run:
for(i in 1:length(threads)){ start(threads[[i]]) }

for(i in 1:length(threads)){ cat("[", i,"] ", state(threads[[i]]), "\n") }


## End(Not run)

## Join the m threads
## Not run:
results <- as.list(1:length(threads))
for(i in 1:length(threads)){ results[[i]] <- join(threads[[i]]) }

print(shared$mu)
print(mean(as.numeric(X), na.rm = TRUE))


## End(Not run)

## Cancel a thread server:
## Not run:
 ThreadServer.cancel(address = attr(server, "address"))


## End(Not run)
```

Index