

Package ‘Rdsm’

February 14, 2012

Version 1.1.0

Author Norm Matloff <normmatloff@gmail.com>

Maintainer Norm Matloff <normmatloff@ugmail.com>

Date 2/1/2011

Title Threads Environment for R

Description Provides a threads-type programming environment for R, usable both on a multicore machine and across a network of multiple machines. The package gives the programmer a shared memory world view, again even across multiple machines on a network.

LazyLoad no

License GPL (>= 2)

Repository CRAN

Date/Publication 2011-02-04 06:44:17

R topics documented:

alinit	2
barr	3
cnewdsm	4
dsmexit	5
dsmv-class	5
fa	6
init	7
lock	7
Rdsm	8
svr	13
wait	14

Index	15
--------------	-----------

alinit

alinit

Description

Initiate and manage an **Rdsm** session.

Usage

```
alinit(ncInt=NULL,nds=NULL,bigmem=F)
cmdtoClnts(cmd)
cmdtoSrvr(cmd)
cmdtoAll(cmd)
go(ncon=2)
loadRdsm(ncInt,bigmem=F)
```

Arguments

ncInt	Number of clients/threads.
nds	A character vector specifying the names of the client nodes.
bigmem	Load bigmemory .
cmd	A one-element character vector specifying a command to run on the clients and/or server.
ncon	Number of clients/threads.

Details

These functions automate the creation and management of an **Rdsm** session. They must be run on a Unix-family system (Linux, Mac OS, or Cygwin on Windows).

Calling `alinit()` creates terminal windows for the server and clients, runs `R` in each of them, loads **Rdsm** and optionally **bigmemory**. Exactly one of `ncInt` and `nds` must be `NULL`, so there are two possibilities:

- `ncInt` specifies the number of clients, all of which run on the local machine, i.e. 'localhost'
- `nds` specifies the client nodes

One initializes the client/server connections by calling `go()`.

The core of automated **Rdsm** session lies in the use of the functions `cmdtoClnts()`, `cmdtoSrvr()`, `cmdtoAll()`, which send the given command to the clients and/or the server.

Here is a sample session, to run a function `x()` contained in the source code file 'y.R':

```
alinit(2) # create 2 clients
cmdtoClnts('source("y.R")') # have clients source the app code
go() # set up server/client connections
cmdtoClnts('x(3,100)') # first run of app
```

```
cmdtoclnts('x(12,5000)') # second run of app
...
```

Note: Autolaunch uses the Unix 'screen' program. If your application crashes, you may need to wipe out leftover screens, both by running `screen -wipe` and killing the associated processes.

Author(s)

Norm Matloff

barr

barr

Description

Classical shared-memory barrier function.

Usage

barr()

Arguments

None.

Details

Call will block at the caller until all clients have made the call. Use to synchronize multiple threads at the same line of code.

Author(s)

Norm Matloff

See Also

[lock](#), [unlock](#), [wait](#), [signal](#), [fa](#)

cnewdsm

cnewdsm

Description

Create a new **Rdsm** or **bigmemory** variable.

Usage

```
cnewdsm(varname, thisclass, thismode=NULL, val)
newdsm(varname, thisclass, thismode=NULL, val=NULL, size=NULL)
newbm(varname, thismode, nr, nc, val=NULL)
```

Arguments

varname	Name of variable to be created.
thisclass	Class of Rdsm variable: "dsmv", "dsmm" or "dsml".
thismode	R mode of variable: NULL for the "dsml" Rdsm case, "integer" or "double" for the other cases; "double" is recommended for bigmemory .
val	Initial value of variable.
size	Size of Rdsm variable: length for a vector, row/col dimension for a matrix.
nr	Number of rows of bigmemory variable.
nc	Number of columns of bigmemory variable.

Details

The functions `cnewdsm()` and `newdsm()` create new **Rdsm** variables, while `newbm()` creates new **bigmemory** variables.

For **Rdsm** variables, one ordinarily calls `cnewdsm()`, resorting to `newdsm()` only if finer control is needed.

In the case of `cnewdsm()`, all clients execute the same call. But if the initial value of the variable is to be that of a non-shared variable in some thread, then that thread calls the function `newdsm()` with a non-NULL `val` but a NULL `size`, while the other clients specify NULL for `val` but a non-NULL for `size`.

For **bigmemory** variables, `val` is not very useful; it's easier to leave it at NULL and then set the variable after the call.

Author(s)

Norm Matloff

`dsmexit`*dsmexit*

Description

Notifies the **Rdsm** server that the client has completed the command sent to it.

Usage

```
dsmexit()  
closecon()
```

Arguments

None.

Details

The server-side function `svrloop()` keeps track of the number of clients remaining. When that number reaches 0, that function returns.

In other words, use of `dsmexit()` eventually results in a shutdown of the server, which then must be restarted if you wish to run further **Rdsm** applications in this session. Thus `dsmexit()` should *not* be used in most applications.

Author(s)

Norm Matloff

See Also

[svr](#)

`dsmv-class`*Rdsm Classes*

Description

The `dsmv`, `dsmm` and `dsm1` classes act like ordinary R vectors, matrices and lists, but have internal operations allowing them to be shared across R processes.

Details

These classes are created by calling `cnewdsm()`. Indexing operations with `[]` work syntactically as with ordinary R vectors, matrices and lists, but the generic R functions `[]` and `[]=` are implemented in special versions to enable the sharing.

(Currently the list functions `[[]]` and `$` are not yet implemented, nor is increasing the number of components of a shared list.)

Note carefully that you must always use brackets with **Rdsm** variables. For instance, to copy the **Rdsm** vector `x` to an ordinary R variable `y`, write

```
y <- x[]
```

not

```
y <- x
```

Author(s)

Norm Matloff

See Also

[cnewdsm](#)

fa

fa

Description

Fetch-and-add function.

Usage

```
fa(fav, inc)
```

Arguments

<code>fav</code>	Fetch-and-add variable (Rdsm integer vector of length 1), name quoted.
<code>inc</code>	Increment value.

Details

When a client calls `fa()` on `fav`, the quantity `inc` (which could be negative) will be added to `fav`, atomically, i.e. without interference by another thread. The actual addition will be performed at the server, thus reducing communications costs.

Author(s)

Norm Matloff

See Also

[barr](#), [wait](#), [signal](#)

init

init

Description

Client calls this to register with the **Rdsm** server, which sends back the client's ID number and the total number of clients.

Usage

```
init(host="localhost",port=2000)
```

Arguments

host	Host IP address
port	Host TCP port number.

Details

This need be run just once per call of `svr()` on the server side. Clients should not rerun `init()` if they wish to retain shared variables.

Author(s)

Norm Matloff

lock

lock

Description

Classical lock, unlock functions.

Usage

```
lock(lockname)  
unlock(lockname)
```

Arguments

lockname Lock variable, quoted.

Details

When a client calls `lock()` on `lockname`, **Rdsm** will check whether `lockname` is locked. If the lock is unlocked, **Rdsm** will lock it, and `lock()` will immediately return. If on the other hand the variable is locked, this client will join the queue for the lock, and its call to `lock()` will block.

When a client calls `unlock()`, the call will immediately return. **Rdsm** will then remove that client from the queue, and will check whether the queue for the lock is still nonempty. If so, then the call to `lock()` made previously by the new head of the queue will now return.

One does not use `newdsm()` to create a lock variable. Instead, the variable is automatically created the first time `lock()` is called on it.

Author(s)

Norm Matloff

See Also

[barr](#), [wait](#), [signal](#), [fa](#)

Rdsm

Threads Programming for R

Description

Rdsm provides a threads programming environment for **R**, not available within **R** itself. Moreover, it is usable both on a multicore machine *and* across a network of multiple machines. The package gives the “look and feel” of the shared memory world view that ordinary system threads provide, again even across multiple machines on a network.

The “dsm” in “Rdsm” stands for *distributed shared memory*, a term from the parallel processing community in which nodes in a cluster share (real or conceptual) memory. It is based on a similar package the author wrote for Perl some years ago (Matloff (2002)).

Rdsm can be used for:

- parallel computation, as with the program ‘KNN.R’ included with this package
- the development of “dashboard” controllers and parallel I/O, like the program ‘WebProbe.R’
- the development of collaborative tools, as with the program ‘Auction.R’

Rdsm can easily be used with variables produced by Jay Emerson and Mike Kane’s **bigmemory** package, thus enhancing the latter package by adding a threads capability. In **bigmemory** case, if the code run on a multicore machine, then the shared memory is real, and the access may be considerably faster than to **Rdsm** variables. **Rdsm** provides a function `newbm()` for creating **bigmemory** variables.

Quick Introduction to Rdsm

The **Rdsm** code in ‘MatMul.R’ in the examples included in this package serves as a quick introduction, using a matrix-multiply example common in parallel processing packages. There are especially detailed comments in this example, but here is an overview:

The code finds the product of matrices *m1* and *m2*, placing the produce in *prd*. The core lines of the code are

```
myid <- myinfo$myid # this thread's ID
# determine number of columns of m1
k <- if(class(m1) == "big.matrix") dim(m1)[2] else m1$size[2]
nth <- myinfo$ncInt # number of threads
chunksize <- k/nth
# determine which columns of m1 this thread will process
firstcol <- 1 + (myid-1) * chunksize
lastcol <- firstcol + chunksize - 1
# process this thread's share of the columns
prd[,firstcol:lastcol] <- m1[,] %*% m2[,firstcol:lastcol]
```

The work is parallelized by assigning each thread a certain set of columns of *prd*. Each thread then computes its columns and then places them in the proper section of *prd*. This is a classical shared-memory pattern, thus illustrating the point that **Rdsm** brings threads programming to **R**.

The matrix *prd* here is a shared variable, created beforehand via a call to `cnewdsm()` in the case of an **Rdsm** variable or via a call to `newbm()` if a **bigmemory** variable is desired.

Other examples, including directions for running them, are given in the ‘examples/’ and ‘testscripts/’ directories in this package.

Advantages of the Shared-Memory Paradigm

Whether the platform is a multicore machine or a set of networked computers, a major advantage of **Rdsm** is that it gives the programmer a shared-memory world view, considered by many in the parallel processing community to be one of the clearest forms of parallel programming (Chandra (2001), Hess *et al* (2003) etc.).

Suppose for instance we wish to copy *x* to *y*. In a message-passing setting such as **Rmpi**, *x* and *y* may reside in processes 2 and 5, say. The programmer would write code (described here in pseudocode)

```
send x to process 5
```

to run on process 2, and write code

```
receive data item from process 2
set y to received item
```

to run on process 5. By contrast, in a shared-memory environment, the programmer would merely write

```
y <- x
```

which is vastly simpler. (Brackets would be required, as explained below.)

This also means that it is easy to convert sequential **R** code to parallel **Rdsm** code.

Packages such as **snw**, arguably in the message-passing realm, do feature more convenient messaging operations, but still shared memory tends to have the simplest code.

(It should be noted, though, that in some applications message-passing can yield somewhat better performance.)

Communication

Rdsm runs via network sockets, and **Rdsm** shared variables are accessed via this mechanism. If one's code also contains **bigmemory** shared variables, these are handled in that package's environment, either physical shared memory or file backing in a shared file system.

Rdsm data communication is binary in the case of vectors and matrices, but `serialize()` and `unserialize()` are used for lists.

Launching Rdsm

Start **R** and load **Rdsm**.

Manual operation:

To run **Rdsm** manually, run **R** in $n+1$ different terminal (shell) windows, where n is the desired number of clients, i.e. degree of parallelism. Each client runs one thread. You will use one of the $n+1$ instances of **R** for the server.

Then:

- Run `srvr()` in your server window, with argument n , which is 2 by default.
- In each client window, run `init()`.
- In each client window, run your **Rdsm** application function.

You may have several application functions to run, or may want to run the same one multiple times. This is fine as long as `srvr()` is still running; you do not need to rerun `init()` at the clients. Application-program **Rdsm** variables etc. will be retained from one run to the next.

Automatic launching:

If you are running on a Unix-family system (Linux, Mac OS, or Cygwin on Windows), **Rdsm** launch and management can be made much more convenient via **Rdsm**'s autolaunch capability. One opens just one window, and autolaunch automatically creates windows for the server and clients, and then in each window starts **R** and loads **Rdsm** (and optionally **bigmemory**).

Then each time the user wishes to issue a command to all the clients, say a command to run an **Rdsm** application, he/she merely types the command in the original window, and it will be sent to the client windows, thus saving a lot of typing.

Here's a quick summary example of autolaunch. Say we wish to run two threads, with our application consisting of a function `x()` contained in the source code file `'y.R'`. We would open a single terminal window, run `R` in it, and then run the following code:

```
alinit(2) # create clients
cmdtoclnts('source("y.R")') # have clients source the app code
go() # set up server/client connections
cmdtoclnts('x(3,100)') # first run of app
cmdtoclnts('x(12,5000)') # second run of app
...
```

Here's what it does:

- The call to `alinit()` opens two other terminal windows, starts `R` in them, and loads the **Rdsm** library.
- The call to `cmdtoclnts()` then has the instances of `R` at the client windows load our application source file.
- The call to `go()` then starts `svr()` in the server window and `init()` in each client window.
- We then run our application a couple of times, and of course could run other **Rdsm** applications after sourcing their code.

Accessing Rdsm Variables

The variables in a typical **Rdsm** application program consist of a few shared variables, produced by either **Rdsm** or **bigmemory**, and many “ordinary” variables. Regular `R` syntax is used to access the shared variables, just as with the ordinary ones.

For example, suppose your program includes `m`, a 4x5 shared matrix variable. If you wished to fill the second column with 1, 2, 3 and 4, you would write

```
m[,2] <- 1:4
```

just as you would in ordinary `R`.

Note carefully that you must always use brackets with shared variables. For instance, to copy the shared vector `x` to an ordinary `R` variable `y`, write

```
y <- x[]
```

not

```
y <- x
```

Built-in Variables

Rdsm's built-in variables are stored in a single global (but not shared) variable `myinfo`, a list consisting of these components:

- `myid`: the ID number of this client, starting with 1
- `nc1nt`: the total number of clients

Built-in Synchronization Functions

Rdsm includes some built-in synchronization functions similar to those of threaded or other shared-memory programming systems:

- `barr()`: barrier operation, synchs all threads to the same code line
- `lock()`: lock operation, gives thread exclusive access to shared variables
- `unlock()`: unlock operation, relinquishes exclusive access
- `wait()`: wait operation
- `signal()`: signal operation; releases all waiting clients
- `signal1()`: same as `signal()`, but releases only the first waiting client
- `fa()`: fetch-and-add operation

Built-in Initialization/Shutdown Functions

- `init()`: initializes a client's connection to the server
- `svr()`: initializes the server
- `dsmexit()`: can be called when a client has finished its work (note: this will stop the server when all clients make this call, and thus this function should not be used in most applications)

Shared-Variable Creation Functions

- `cnewdsm()`: creates an **Rdsm** variable
- `newbm()`: creates a **bigmemory** variable

Internal Structure

Though transparent to the **Rdsm** programmer, internally **Rdsm** variables (but not **bigmemory** ones) have the following architecture.

The **Rdsm** application variables reside on the server. Each read from or write to an **Rdsm** variable involves a transaction with the server. **Rdsm** variables reference vectors, matrices and lists, but have the special **Rdsm** classes `dsmv`, `dsmm` and `dsm1`, respectively. Indexing operations for these classes communicate with the server to read or write the desired objects.

See the **bigmemory** package for details of the structure used for those variables. These are of the matrix type only, class `big.matrix`. Of course, a vector can be represented as a one-row vector.

Again, all this is transparent to the programmer. However, as with any system, a good understanding of the internals can result in your writing much better code.

Author(s)

Norm Matloff

References

Chandra, Rohit (2001), *Parallel Programming in OpenMP*, Kaufmann, pp.10ff (especially Table 1.1).

Hess, Matthias *et al* (2003), Experiences Using OpenMP Based on Compiler Directive Software DSM on a PC Cluster, in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools*, Michael Voss (ed.), Springer, p.216.

Matloff, Norman (2002), PerlDSM: A Distributed Shared Memory System for Perl. *Proceedings of PDPTA 2002*, 2002, 63-68.

svr	svrinit
-----	---------

Description

Server-side code. Invocation of the **Rdsm** server.

Usage

```
svrinit(port=2000,ncon=2)
svrloop()
svr(port=2000,ncon=2)
```

Arguments

port	Server TCP port number
ncon	Number of connections, i.e. number of clients.

Details

The function `svrinit()` accepts client requests for connection with the server, and returns to each client its ID and the total number of clients. Then `svrloop()` is called to run the server.

Then, at each client, `init()` must be called.

One normally uses `svr()` in lieu of the pair `svrinit()` and `svrloop()`.

You may have several application functions to run, or may want to run the same one multiple times. This is fine as long as you don't rerun `svrinit()`; run `svr()` just once on the server, and run `init()` just once at the clients. Application-program **Rdsm** variables etc. will be retained from one run to the next.

Author(s)

Norm Matloff

See Also

[init](#)

wait

wait

Description

Threads-like wait, signal

Usage

```
wait(waitvarname)
signal(waitvarname)
signal1(waitvarname)
```

Arguments

waitvarname Wait variable, quoted.

Details

When a client calls `wait()` on the wait variable `waitvarname()`, **Rdsm** will add this client's ID to a list for that variable. The call will block. When another client later calls `signal()`, **Rdsm** will then release all the clients, i.e. their calls to `wait()` will unblock. If you wish to release just the first waiting client, use `signal1()`.

One does not use `newdsm` to create a wait variable. Instead, the variable is automatically created the first time `wait()` is called on it.

A signal posted when no waits are pending will be ignored.

Author(s)

Norm Matloff

See Also

[barr](#), [lock](#), [unlock](#), [fa](#)

Index

[.dsm1 (dsmv-class), 5
[.dsmm (dsmv-class), 5
[.dsmv (dsmv-class), 5
[<-.dsm1 (dsmv-class), 5
[<-.dsmm (dsmv-class), 5
[<-.dsmv (dsmv-class), 5

alinit, 2

barr, 3, 7, 8, 14

closecon (dsmexit), 5
cmdtoall (alinit), 2
cmdtoclnts (alinit), 2
cmdtosrvr (alinit), 2
cnewdsm, 4, 6

dsmexit, 5
dsmv-class, 5

fa, 3, 6, 8, 14

go (alinit), 2

init, 7, 13

loadrdsm (alinit), 2
lock, 3, 7, 14

newbm (cnewdsm), 4
newdsm (cnewdsm), 4

Rdsm, 8

signal, 3, 7, 8
signal (wait), 14
signal1 (wait), 14
srvr, 5, 13
srvrinit (srvr), 13
srvrloop (srvr), 13

unlock, 3, 14
unlock (lock), 7

wait, 3, 7, 8, 14