Package 'memshare'

December 5, 2025

Type Package

Title Shared Memory Multithreading

Version 1.1.0

Date 2025-12-05

Description This project extends 'R' with a mechanism for efficient parallel data access by utilizing 'C++' shared memory. Large data objects can be accessed and manipulated directly from 'R' without redundant copying, providing both speed and memory efficiency.

Maintainer Michael Thrun <m.thrun@gmx.net>

LazyLoad yes

LinkingTo Rcpp

Imports Rcpp (>= 1.0.14), parallel

Suggests ScatterDensity (>= 0.1.1), DataVisualizations (>= 1.1.5), mpmi, rmarkdown (>= 0.9), knitr (>= 1.12), testthat (>= 3.0.0)

Config/testthat/edition 3

SystemRequirements C++17

Depends R (>= 4.3.0)

NeedsCompilation yes

License GPL-3

URL https://www.iap-gmbh.de

Encoding UTF-8

VignetteBuilder knitr

BugReports https://github.com/Mthrun/memshare/issues

Author Julian Maerte [aut, ctr] (ORCID: <https://orcid.org/0000-0001-5451-1023>), Romain Francois [ctb], Michael Thrun [aut, ths, rev, cph, cre] (ORCID: <https://orcid.org/0000-0001-9542-5543>)

Repository CRAN

Date/Publication 2025-12-05 08:00:02 UTC

Contents

`memshare-package` *Shared Memory Multithreading*

Description

This project extends 'R' with a mechanism for efficient parallel data access by utilizing 'C++' shared memory. Large data objects can be accessed and manipulated directly from 'R' without redundant copying, providing both speed and memory efficiency.

Details

The DESCRIPTION file:

Package:	memshare
Type:	Package
Title:	Shared Memory Multithreading
Version:	1.1.0
Date:	2025-12-05
Authors@R:	c(person("Julian" ,"Maerte",email= "j.maerte@iap-gmbh.de",role=c("aut","ctr"), comment = c(ORC
Description:	This project extends 'R' with a mechanism for efficient parallel data access by utilizing 'C++' share
Maintainer:	Michael Thrun <m.thrun@gmx.net>
LazyLoad:	yes
LinkingTo:	Rcpp
Imports:	Rcpp (>= 1.0.14), parallel
Suggests:	ScatterDensity (>= 0.1.1), DataVisualizations (>= 1.1.5), mpmi, rmarkdown (>= 0.9), knitr (>= 1.12
Config/testthat/edition:	3
SystemRequirements:	C++17
Depends:	R (>= 4.3.0)
NeedsCompilation:	yes
License:	GPL-3
URL:	https://www.iap-gmbh.de

Encoding:	UTF-8
VignetteBuilder:	knitr
BugReports:	https://github.com/Mthrun/memshare/issues
Author:	Julian Maerte [aut, ctr] (ORCID: <https://orcid.org/0000-0001-5451-1023>), Romain Francois [ctb]

Index: This package was not yet installed at build time.

If the user detaches the package, all handels are destroyed, meaning that all vairables of all namespaces are cleared as long as there is no other r thread still using the variables.

The two basic definitions are:

1. "`Pages`" are variables owned by the current compilation unit of the code (e.g., 'R' session or terminal that loaded the DLL). The pages are coded in Windows via 'MapViewOfFile' and on Unix via 'shm'+'mmap'.

2. "`Views`" are references to variables owned by another (or their own) compilation unit. The views are always 'ALTREP' wrappers for the pointers to the shared memory chunk.

3. "`namespace`" are character of length 1 called here strings, that define the identifier of the shared memory context allowing the initialize shared variables.

Safety

R itself is designed around a single-threaded C API, which means that internal R functions and memory management cannot be called safely from multiple threads at the same time. Each worker process created by **parallel** runs its own independent R interpreter, so ordinary R code is safe as long as all R-level operations happen inside one worker at a time.

However, shared-memory buffers created by **memshare** are visible to multiple R sessions simultaneously. These buffers are intended to be *read-shared*: many workers can read the same matrix or vector concurrently without conflict. If you ever modify a shared object in place (e.g., `X[1,1] <- 0`), that write immediately affects the shared buffer seen by other workers and can cause data races if they also access the same region. To avoid this, treat shared variables as read-only, or implement explicit synchronization mechanisms such as interprocess locks or task partitioning that guarantees non-overlapping writes.

Inside compiled code (e.g., via Rcpp, OpenMP, or TBB), threads may perform numerical computations on raw pointers to shared data, but must never call back into R (e.g., create R objects, print, evaluate R expressions, etc.) from those secondary threads. All communication with R must occur in the main thread of each worker process.

Resource lifecycle

Shared memory behaves differently from ordinary R objects because it exists outside the R garbage collector. Therefore, **memshare** provides explicit functions to manage it.

Each time you call `retrieveViews`, the package creates one or more "handles" that link your R objects to the shared-memory segments. When you are done using these views, always call `releaseViews` to remove those handles. As long as any active view exists in any R session, the corresponding shared memory remains allocated.

Memory can only be reclaimed once all views have been released. The call to `releaseVariables` from the master session removes ownership of the pages, but they are physically unmapped only when no process holds a view. Detaching or unloading the **memshare** package automatically drops

all handles, but if another R session still has an open view, that memory remains in use until it is released there as well. This design prevents dangling pointers and ensures that shared data are not invalidated while still in use by another worker.

OS notes

Windows. Uses Win32 file mappings (`CreateFileMappingA()`, `MapViewOfFile()`). Namespaces are automatically prefixed with `"Local\\"` to scope mappings per user session. Mapping sizes use 64-bit high/low DWORDs. Views are opened read-only by default (`FILE_MAP_READ`). Cleanup unmaps views (`UnmapViewOfFile()`) and then closes handles (`CloseHandle()`). Requires homogeneous architecture (e.g., all 64-bit R sessions). Antivirus/EDR tools may slow mappings.

Linux. Uses POSIX shared memory (`shm_open()` + `mmap()`). Owners create with `O_CREAT|O_EXCL|O_RDWR` and `PROT_READ|PROT_WRITE`; views attach with `O_RDONLY` and `PROT_READ`. Owners call `shm_unlink()` only after all views are released. Shared memory size may be limited by `/dev/shm` or system SHM limits (e.g., `shmmax`).

macOS. Also uses POSIX shared memory (`shm_open()` + `mmap()`) with the same owner/view flags as Linux. Unlike Linux, macOS implements POSIX shared memory via files in '`/var/run/shm`' or '`/private/var/shm`', which may not exist by default and can require manual creation or permission adjustment. POSIX shm names must be short; a practical upper bound is about 32 characters including the leading slash. Exceeding this limit raises a clear error. Shared memory segments are visible only to processes of the same user unless permissions are relaxed. The maximum segment size could be smaller than on Linux. macOS does not automatically remove orphaned segments if a process crashes; these must be manually cleaned using `ipcs -m` / `ipcrm` or by restarting the system. If your R session or script crashes before releaseVariables() or releaseViews() runs, those shared memory segments may remain allocated on disk.

Author(s)

Julian Maerte [aut, ctr] (ORCID: <https://orcid.org/0000-0001-5451-1023>), Romain Francois [ctb], Michael Thrun [aut, ths, rev, cph, cre] (ORCID: <https://orcid.org/0000-0001-9542-5543>)

Maintainer: Michael Thrun <m.thrun@gmx.net>

Examples

```
x = rnorm(100)
y = runif(100)
Mat = cbind(x,x,x)
res = memApply(X = Mat, MARGIN = 2,
FUN = function(x,y) {
  cc = memshare::mutualinfo(x,y,isYDiscrete = TRUE,
                      na.rm = TRUE,useMPMI = FALSE)
  return(cc)
},VARS = list(y=y),MAX.CORES=1, #for testing purposes only single thread
NAMESPACE = "namespaceID")
unlist(res)
## Not run:
#usually MAX.CORES>1 for application

#alternative usage with manual memory allocation:

## End(Not run)
```

```
Data = cbind(x, x, x)
namespace = "ns_package"
memshare::registerVariables(namespace, list(Data = Data, y = y))
res2 = memshare::memApply(
  X = "Data",
  MARGIN = 2,
  FUN = function(x, y) {
    cc = memshare::mutualinfo(x,
                              y,
                              isYDiscrete = TRUE,
                              na.rm = TRUE,
                              useMPMI = FALSE)
    return(cc)
  },
  VARS = c("y"),
  MAX.CORES = 1,
  #for testing purposes only single thread
  NAMESPACE = namespace
)
unlist(res2)
memshare::releaseVariables(namespace, c("Data", "y"))
```

---

memApply *Analog of* parApply *function for a shared memory context.*

Description

memApply mirrors parApply in the shared memory setting given a shared memory space namespace with a target matrix X and some shared variables VARS either as variables or as names of their registered variables.

Usage

```
memApply(X, MARGIN, FUN,

NAMESPACE = NULL, CLUSTER=NULL, VARS=NULL, MAX.CORES=NULL)
```

Arguments

X	A [1:n,1:d] numerical matrix of n rows and d columns which is worked upon. Can also be a string name of an already registered variable in NAMESPACE; otherwise will be registered automatically.
MARGIN	Whether to apply by row (1) or column (2).
FUN	Function that is applied on either the rows or columns of X. The first argument will be set to the vector and the subsequent arguments have to have the same name as their registered variables.
NAMESPACE	Optional, string. The namespace identifier for the shared memory session. If this is NULL it will be set to the name of FUN in runtime environment. However for inline-defined functions FUN an explicit NAMESPACE is recommended.

CLUSTER	Optional, A parallel::makeCluster cluster. Will be used for parallelization. By defining clusterExport constant R-copied objects (non-shared) can be shared among different executions of FUN. If NULL we initialize a new one.
VARS	Optional, Either a named list of variables where the name will be the name under which the variable is registered in shared memory space or a character vector of names of variables already registered which should be provided to FUN.
MAX.CORES	Optional, In case CLUSTER is undefined a new cluster with MAX.CORES many cores will be initialized. If NULL we use detectCores() - 1 many.

Details

memApply runs a worker pool on the exact same memory (for shared memory context, see registerVariables), and allows you to apply a function FUN row- or columnwise (depending on MARGIN) over the target matrix. Since the memory is shared only the names of variables have to be copied to each worker thread in CLUSTER (a makeCluster multithreading cluster) resulting in sharing of arbitrarily large matrices (as long as the fit in RAM once) along a **parallel** cluster while only copying a couple of bytes per cluster.

The numerical matrix X and the Vars havee to be objects of base type 'double'.

It is recommended not to change the values of v inside FUN, however this will only lead to some copying of the column whenever it is worked upon; the shared memory thus will not be corrupted even if you write to column or row. Also the copying only ever happens for one column/row at a time leading to much lower memory consumption than parallel even in this case.

Thread safety

The vector v passed to FUN is typically an ALTREP view that *directly references shared memory* rather than a private copy. This means that multiple worker processes may be reading the same memory region simultaneously.

Read-only operations are fully safe and recommended. Examples include statistical summaries (mean(v), cor(v, y)), vectorized arithmetic, and model-fitting that does not modify v.

If you attempt to modify elements of v directly (for example, v[1] <- 0), you are writing into a shared buffer. Concurrent modification by multiple workers can lead to race conditions or data corruption. Even if no other process is writing, in-place assignment may still trigger an internal copy of that row or column, slightly increasing memory usage.

For safety and clarity, always *copy* v locally if you need to modify it:

```
f <- function(v, y) {
  v <- as.vector(v)  # make a private, normal R copy
  v <- scale(v)
  cor(v, y)
}
```

This ensures isolation between workers and prevents unintended data sharing.

Finally, remember that R's internal C API is not thread-safe. If your function FUN uses multi-threaded C++ code (e.g., via OpenMP or TBB), those internal threads must *not* make calls into R (such as creating objects, evaluating expressions, or printing). All R interactions must occur in the main thread of each worker process.

Value

`result` A list of the results of func(row,...) of size n or func(col, ...) of size d, depending on `MARGIN`, for every row/col of `X`.

Author(s)

Julian Maerte

See Also

`parApply`

Examples

```
library(parallel)
cl = makeCluster(1)
i = 1
A1 = matrix(as.double(1:10^(i+1)),10^i, 10^i)

res = memApply(X = A1, MARGIN = 2, FUN = function(x) {
  return(sd(x))
}, CLUSTER=cl, NAMESPACE="ns_apply")

SD_vector=unlist(res)
```

---

memLapply *Analog of* parLapply *function for a shared memory context.*

Description

`memLapply` mirrors `parLapply` in the shared memory setting given a shared memory space `namespace` with a target list `X` and some shared variables `VARS` either as list of variables or as their names in the memory space,

Usage

```
memLapply(X, FUN,

NAMESPACE = NULL, CLUSTER = NULL, VARS=NULL, MAX.CORES = NULL)
```

Arguments

`X`	Either a 1:n list object or a the name of an already registered list object in `NAMESPACE`.
`FUN`	Function to be applied over the list. The first argument will be set to the list element, the remaining ones have to have the same name as they have in the shared memory space!

NAMESPACE	Optional, string. The namespace identifier for the shared memory session. If this is NULL it will be set to the name of FUN in runtime environment. However for inline-defined functions FUN an explicit NAMESPACE is recommended.
CLUSTER	Optional, A parallel::makeCluster cluster. Will be used for parallelization. By defining clusterExport constant R-copied objects (non-shared) can be shared among different executions of FUN. If NULL we initialize a new one.
VARS	Optional, Either a named list of variables where the name will be the name under which the variable is registered in shared memory space or a character vector of names of variables already registered which should be provided to FUN.
MAX.CORES	Optional, In case CLUSTER is undefined a new cluster with MAX.CORES many cores will be initialized. If NULL we use `detectCores() - 1` many.

Details

memLapply runs a worker pool on the exact same memory (shared memory context), and allows you to apply a function FUN elementwise over the target list. Since the memory is shared only the names have to be copied to each worker thread in CLUSTER (a makeCluster multithreading cluster) resulting in sharing of arbitrarily large matrices (as long as the fit in RAM once) along a **parallel** cluster while only copying a couple of bytes per cluster. It is recommended not to change the values of the list element el inside FUN, however this will only lead to some copying of the element whenever it is worked upon; the shared memory thus will not be corrupted even if you write to an element. Also the copying only ever happens for one element at a time leading to much lower memory consumption than parallel even in this case.

Thread safety

Each element el provided to FUN is typically an ALTREP view of a shared-memory object rather than an ordinary R copy. This means that workers can access the same physical memory region concurrently.

Read-only operations are fully safe and recommended. Examples include computations such as matrix multiplication, summary statistics, or transformations that return new results without altering el in place. Because the data are shared, these operations require almost no additional memory and avoid costly data duplication.

If your function modifies an element directly (e.g., `el[1,1] <- 0`), you may be writing to a shared memory buffer that other workers can also access. This can cause inconsistent results or data corruption if multiple workers write simultaneously. Even when no overlap exists, such in-place modification can trigger a copy of the element inside that worker, reducing memory efficiency.

For safety, make an explicit local copy before any in-place changes:

```
f <- function(el, y) {
  el <- as.matrix(el)  # force a private copy
  el <- el * y         # modify locally
  colSums(el)
}
```

This ensures that only the current worker modifies its own private memory.

Finally, note that R's internal C API is single-threaded. If FUN calls compiled code (e.g., via Rcpp, OpenMP, or TBB) that spawns multiple threads, those threads must not interact directly with R

(e.g., by creating R objects, printing, or evaluating expressions). All such interactions must occur in the main R thread of each worker process.

Value

`result` A 1:n list of the results of func(list[[i]],...), for every element of listName.

Author(s)

Julian Maerte

See Also

parLapply

Examples

```
list_length = 1000
matrix_dim = 100

l = lapply(
    1:list_length,
    function(i) matrix(rnorm(matrix_dim * matrix_dim),
    nrow = matrix_dim, ncol = matrix_dim))

y = rnorm(matrix_dim)

namespace = "ns_lapply"
res = memshare::memLapply(l, function(el, y) {
  el
}, NAMESPACE=namespace, VARS=list(y=y), MAX.CORES = 1)
```

---

memshare_gc *Function to remove all handles (ownership and viewership) for a namespace in a worker context.*

Description

Given a namespace identifier (identifies the shared memory space to register to), this function removes all handles to shared memory held by the master and a worker context.

Usage

```
memshare_gc(namespace, cluster)
```

Arguments

`namespace`	string of the identifier of the shared memory context.
`cluster`	A worker context (parallel cluster) that holds views or pages in the same memory context as the master. NULL by default; then only the master session gets its handles removed.

Value

No return value, called deallocation of memory pages and views in a joint memory context.

Author(s)

Julian Maerte

See Also

releaseVariables, releaseViews

Examples

```
  cluster = parallel::makeCluster(1)

  mat = matrix(0,5,5)
  registerVariables("ns", list(mat=mat))

  parallel::clusterEvalQ(cluster, {
    view = memshare::retrieveViews("ns", c("mat"))
  })
  ## Not run:
  # At this point each worker holds a view of mat

## End(Not run)
  memshare_gc("ns", cluster)
  ## Not run:
  # Every workers viewership handle gets destroyed, master sessions page handle
  # gets destroyed.
  # As no handles are left open, the memory is free'd.

## End(Not run)
  parallel::stopCluster(cluster)
```

---

mutualinfo *Mutual Information of continuous and discrete variables.*

Description

Return mutual information for a pair of joint variables. The variables can either be both numeric, both discrete or a mixture. The calculation is done via density estimate whenever necessary (i.e. for the continuous variables). The density is estimated via pareto density estimation with subsequent gaussian kernel smoothing.

Usage

```
mutualinfo(x, y, isXDiscrete = FALSE, isYDiscrete = FALSE,

eps=.Machine$double.eps*1000, useMPMI=FALSE,na.rm=FALSE)
```

Arguments

x	[1:n] a numeric vector (not necessarily continuous)
y	[1:n] a numeric vector (not necessarily continuous)
isXDiscrete	Boolean defining whether or not the first numeric vector resembles a continuous or discrete measurement
isYDiscrete	Boolean defining whether or not the second numeric vector resembles a continuous or discrete measurement
eps	Scalar, The threshold for which the mutual info summand should be ignored (the limit of the summand for x -> 0 is 0 but the logarithm will be -inf...)
useMPMI	Boolean defining whether or not to use the package **mpmi** for the calculation (will be used as a baseline)
na.rm	Boolean defining whether or not to use complete obeservations only

Details

Mutual Information is >= 0 and symmetric (in x and y). You can think of mutual information as a measure of how much of x's information is contained in y's information or put more simply: How much does y predict x. Note that mutual information can be compared for pairs that share one variable e.g. (x,y) and (y,z), if MI(x,y) > MI(y,z) then x and y are more closely linked than y and z. However given pairs that do not share a variable, e.g. (x,y), (u,v) then MI(x,y) and MI(u,v) can not be reasonably compared. In particular: MI defines a partial ordering on the column pairs of a matrix instead of a total ordering (which correlation does for example). This is mainly due to MI not being upper-bound and thus is not reasonable put on a scale from 0 to 1.

Value

mutualinfo	The mutual information of the variables

Note

This function requires that either **DataVisualizations** and **ScatterDensity** of equal or higher version than `0.1.1` is installed, or **mpmi** package

Author(s)

Julian Märte, Michael Thrun

References

Claude E. Shannon: A Mathematical Theory of Communication, 1948

Examples

```
x = c(rnorm(1000),rnorm(2000)+8,rnorm(1000)*2-8)
y = c(rep(1, 1000), rep(2, 2000), rep(3,1000))


if(requireNamespace("DataVisualizations", quietly = TRUE) &&
```

```
   requireNamespace("ScatterDensity", quietly = TRUE) &&
   packageVersion("ScatterDensity") >= "0.1.1" &&
    packageVersion("DataVisualizations") >= "1.1.5"){

  mutualinfo(x, y, isXDiscrete=FALSE, isYDiscrete=TRUE)
}


  if(requireNamespace("mpmi", quietly = TRUE)) {

  mutualinfo(x, y, isXDiscrete=FALSE, isYDiscrete=TRUE,useMPMI=TRUE)
  }
```

`pageList` *Function to obtain a list of the registered variables of the current session.*

Description

When your current session has registered shared memory variables via `registerVariables` internally the variable is tracked until it is released via `releaseVariables`.

This function serves as a tool to check whether all variables have been free'd after usage or to see what variables are currently held by the session.

Usage

```
pageList()
```

Details

The string of each element of the output list has the format environment, backslash, backslash <namespace name>.<variable name>. Default is lokal environment.

Value

An [1:m] list of characters of the registered p namespaces, each of them having up to k variables, m<=p*k. Each element of the list is a combination of namespace and variable name

Note

Use alongside `viewList` to ensure all views and pages are cleared before shutdown.

Author(s)

Julian Maerte

See Also

`viewList`, `registerVariables`, `releaseVariables`

Examples

```
  pageList()
  ## Not run:
  # = list()

## End(Not run)
  mat = matrix(0,5,5)
  registerVariables("ns_pageL", list(mat=mat))
  pageList()
  ## Not run:
  # = list("mat")

## End(Not run)
  releaseVariables("ns_pageL", c("mat"))
  pageList()
  ## Not run:
  # = list()

## End(Not run)
```

`registerVariables` *Function to register variables in a shared memory space.*

Description

Given a namespace identifier (identifies the shared memory space to register to), this function allows you to allocate shared memory and copy data into it for other R sessions to access it.

Usage

```
registerVariables(namespace, variableList)
```

Arguments

`namespace`	string of the identifier of the shared memory context.
`variableList`	A named list of variables to register. Currently supported are matrices and vectors.

Value

No return value, called for allocation of memory pages.

Author(s)

Julian Maerte

See Also

`releaseVariables`, `retrieveViews`

Examples

```
library(memshare)
n = 10
m = 10

TargetMat= matrix(rnorm(n * m), n, m) # target matrix
x_vec = rnorm(n) # some other vector

namespace = "ns_register"
registerVariables(namespace, list(TargetMat=TargetMat, x_vec=x_vec))
memshare::releaseVariables(namespace, c("TargetMat", "x_vec"))
```

`releaseVariables` *Release variables from a shared memory namespace*

Description

Delete variables from the shared memory space. Actual release occurs only when no active views remain.

Usage

```
releaseVariables(namespace, variableNames)
```

Arguments

`namespace`	Character(1) used at registration time.
`variableNames`	Character vector of names to free.

Details

Registered buffers remain allocated until *all* views are released. If any worker still holds a view, `releaseVariables` cannot reclaim memory.

Thread safety

Registered buffers may be read concurrently by many processes. Concurrent writes must be synchronized externally (e.g., interprocess mutex). Do not call the R API from secondary threads.

Wrappers such as `memApply` / `memLapply` call this function on.exit.

Value

Invisibly, `TRUE` on success.

Note

This call succeeds in removing ownership, but underlying memory is only unmapped when every process has called `releaseViews` for those variables. Use `viewList` and `pageList` for diagnostics.

See Also

registerVariables, releaseViews

Examples

```
ns <- "example"
X <- matrix(rnorm(100), 10, 10)
registerVariables(ns, list(X = X))
# later ...
releaseVariables(ns, "X")
```

---

releaseViews *Function to release views of a shared memory space.*

Description

Given a namespace identifier (identifies the shared memory space to register to), this function releases retrieved views from the shared memory space.

NOTE: All views have to be free'd upon releasing the variable by the master.

Usage

```
releaseViews(namespace, variableNames)
```

Arguments

namespace	string of the identifier of the shared memory context.
variableNames	A character vector of variable names to delete.

Details

This is the only way to drop handles created by retrieveViews. Wrappers such as memApply / memLapply call this function internaly. Not releasing views effectively leaks the backing pages for the lifetime of the process.

Value

No return value, called for deallocation of views.

Author(s)

Julian Maerte

See Also

retrieveViews, registerVariables

Examples

```
  ## Not run:
  # MASTER SESSION:
  # allocate data

## End(Not run)
  n = 1000
  m = 100

  mat = matrix(rnorm(n * m), n, m) # target matrix
  y = rnorm(n) # some other constant vector in which the function should not run

  namespace = "ns_relview"
  memshare::registerVariables(namespace, list(mat=mat, y=y))
  ## Not run:
  # WORKER SESSION:

## End(Not run)
  res = retrieveViews(namespace, c("mat", "y"))
  ## Not run:
  # Perform your shared calculations here

## End(Not run)
  releaseViews(namespace, c("mat", "y"))
  ## Not run:
  # MASTER SESSION:
  # free memory

## End(Not run)
  memshare::releaseVariables(namespace, c("mat", "y"))
```

`retrieveMetadata` *Function to obtain the metadata of a variable from a shared memory space.*

Description

Given a namespace identifier (identifies the shared memory space to register to), this function retrieves the metadata of the stored variable.

NOTE: If no view of the variable was previously retrieved this implicitly retrieves a view and thus has to free'd afterwards!

Usage

```
retrieveMetadata(namespace, variableName)
```

Arguments

namespace	string of the identifier of the shared memory context.
variableName	[1:m] character vector, names of one ore more than one variable to retrieve the metadata from the shared memory space.

Details

In some contexts, querying metadata may create an implicit view. If so, you must call releaseViews for that variable afterwards. See examples.

Value

A [1:m] named list mapping the variable names to their retrieved metadata. Each list element contains a list of two elements called "type" and length "n"

Author(s)

Julian Maerte

See Also

releaseVariables, releaseViews, registerVariables

Examples

```
  ## Not run:
  # MASTER SESSION:
  # allocate data

## End(Not run)
  n = 1000
  m = 100

  mat = matrix(rnorm(n * m), n, m) # target matrix

  namespace = "ns_meta"
  memshare::registerVariables(namespace, list(mat=mat))
  ## Not run:
  # WORKER SESSION:
  # retrieve metadata of the variable

## End(Not run)
  res = memshare::retrieveMetadata(namespace, "mat")
  ## Not run:
  # res$type = "matrix"
  # res$nrow = 1000
  # res$ncol = 100

## End(Not run)
  releaseViews(namespace, c("mat"))
  ## Not run:
```

```
  # MASTER SESSION:
  # free memory

## End(Not run)
  memshare::releaseVariables(namespace, c("mat"))
```

`retrieveViews` *Function to obtain an 'ALTREP' representation of variables from a shared memory space.*

Description

Given a namespace identifier (identifies the shared memory space to register to), this function constructs mocked matrices/vectors (depending on the variable type) pointing to 'C++' shared memory instead of 'R'-internal memory state. The mockup is constructed as an 'ALTREP' object, which is an **Rcpp** wrapper around 'C++' raw memory. 'R' thinks of these objects as common matrices or vectors.

The variables content can be modified, resulting in modification of shared memory. Thus when not using wrapper functions like `memApply` or `memLapply` the user has to be cautious of the side-effects an 'R' session working on shared memory has on other 'R' sessions working on the same namespace.

Usage

```
retrieveViews(namespace, variableNames)
```

Arguments

`namespace`	string of the identifier of the shared memory context.
`variableNames`	[1:n] character vector, the names of the variables to retrieve from the shared memory space.

Details

Thread safety

Returned objects may alias shared memory. Concurrent writes must be synchronized externally (e.g., interprocess mutex). Do not call the R API from secondary threads.

Resource cleanup

Each call must be matched by `releaseViews`. Failing to release views prevents `releaseVariables` from freeing memory.

Value

An 1:p list of p elements, each element contains a variable that was registered by `registerVariables`

Note

Having a view of a memory chunk introduces an internally tracked handle to the shared memory. Shared memory is not deleted until all handles are gone; before calling releaseVariables in the master session, you have to free all view-initialized handles via releaseViews!

Author(s)

Julian Maerte

See Also

releaseVariables, registerVariables, releaseViews

Examples

```
  ## Not run:
  # MASTER SESSION:
  # init some data and make shared

## End(Not run)
  n = 1000
  m = 100

  mat = matrix(rnorm(n * m), n, m) # target matrix
  y = rnorm(n) # some other constant vector in which the function should not run

  namespace = "ns_retrview"
  memshare::registerVariables(namespace, list(mat=mat, y=y))
  ## Not run:
  # WORKER SESSION
  # retrieve the shared data and work with it

## End(Not run)
  res = memshare::retrieveViews(namespace, c("mat", "y"))
  ## Not run:
  # res is a list of the format:
  # list(mat=matrix_altrep, y=vector_altrep),
  # altrep-variables can be used
  # exactly the same way as a matrix or vector
  # and also behave like them when checking via
  # is.matrix or is.numeric.


  # important: Free view before resuming
  # to master session to release the variables!

## End(Not run)
  memshare::releaseViews(namespace, c("mat", "y"))

  ## Not run:
  # MASTER SESSION
  # After all view handles have been free'd, release the variable
```

```
## End(Not run)
  memshare::releaseVariables(namespace, c("mat", "y"))
```

`viewList` *Function to obtain a list of the views the current session holds.*

Description

When your current session has retrieved views of shared memory via `retrieveViews` internally the view is tracked until it is released via `releaseViews`.

This function serves as a tool to check whether all views have been free'd after usage or to see what views are currently available to the session.

Usage

```
viewList()
```

Details

The string of each element of the output list has the format <namespace name>.<variable name>. Default is lokal environment.

Useful for *leak diagnostics*: if `releaseVariables` did not free memory, check that `viewList()` is empty across all sessions.

Value

An 1:p list of characters of the the p retrieved views

Note

For windows we prepend the namespace identifier by "`Local\\`" because otherwise the shared memory is shared system-wide (instead of user-wide) which needs admin privileges.

Author(s)

Julian Maerte

See Also

`retrieveViews`, `releaseViews`

Examples

```
  ## Not run:
  # MASTER SESSION:

## End(Not run)
  mat = matrix(0,5,5)
  registerVariables("ns_viewL", list(mat=mat))

  ## Not run:
  # WORKER SESSION:

## End(Not run)
  viewList() # an empty list to begin with (no views retrieved)

  matref = retrieveViews("ns_viewL", c("mat"))
  viewList()
  ## Not run: # now equals c("ns_viewL.mat")

  releaseViews("ns_viewL", c("mat"))
  viewList()
  ## Not run:
  # an empty list again

  # MASTER SESSION:

## End(Not run)
  releaseVariables("ns_viewL", c("mat"))
```

Index