

# Package ‘rpvm’

January 2, 2012

**Version** 1.0-4

**Date** 2010-04-16

**Title** R interface to PVM (Parallel Virtual Machine)

**Author** Na (Michael) Li <wuolong@gmail.com> and A. J. Rossini <rossini@u.washington.edu>.

**Maintainer** Na (Michael) Li <wuolong@gmail.com>

**Depends** R (>= 2.0.0)

**Suggests** rsprng

**Description** Provides interface to PVM APIs, and examples and documentation for its use.

**License** GPL (>= 2)

**URL** <http://www.r-project.org/>

**Repository** CRAN

**Date/Publication** 2010-04-17 18:38:40

## R topics documented:

.PVM.buinfo . . . . .	2
.PVM.config . . . . .	3
.PVM.exit . . . . .	4
.PVM.freebuf . . . . .	5
.PVM.initsend . . . . .	6
.PVM.kill . . . . .	7
.PVM.mcast . . . . .	8
.PVM.mkbuf . . . . .	9
.PVM.mstats . . . . .	10
.PVM.mytid . . . . .	11
.PVM.notify . . . . .	12
.PVM.nrecv . . . . .	13
.PVM.probe . . . . .	14

.PVM.pstats . . . . .	15
.PVM.recv . . . . .	16
.PVM.send . . . . .	17
.PVM.spawn . . . . .	19
.PVM.tasks . . . . .	20
.PVM.tidtohost . . . . .	22
.PVM.trecv . . . . .	22
init.sprng.master . . . . .	24
PVM.barrier . . . . .	25
PVM.bcast . . . . .	26
PVM.buffers . . . . .	27
PVM.gather . . . . .	28
PVM.getinst . . . . .	30
PVM.gettid . . . . .	31
PVM.group . . . . .	32
PVM.gsize . . . . .	33
PVM.options . . . . .	34
PVM.pack . . . . .	36
PVM.rapply . . . . .	38
PVM.reduce . . . . .	38
PVM.scatter . . . . .	40
PVM.serialize . . . . .	41
PVM.unpack . . . . .	42
PVMD . . . . .	44

<b>Index</b>	<b>46</b>
--------------	-----------

---

<i>.PVM.bufinfo</i>	<i>Message buffer infomation</i>
---------------------	----------------------------------

---

## Description

Returns information about the requested message buffer.

## Usage

`.PVM.bufinfo (bufid)`

## Arguments

bufid	id of a particular message buffer
-------	-----------------------------------

## Details

Returns information about the requested message buffer. Typically it is used to determine about the last received message such as it size or source. It is especially useful when an application is able to receive any incoming message (with the use of wildcards).

**Value**

Return a list (bytes, msgtag, tid) of information about the requested message buffer.

bytes            the length in bytes of the entire message  
msgtag           the message label  
tid               the source of the message

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.send](#), [.PVM.nrecv](#), [.PVM.recv](#)

**Examples**

```
## Not run: bufid <- .PVM.recv (-1, -1)  
## Not run: info <- .PVM.bufinfo (bufid)
```

---

.PVM.config	<i>PVMD configuration</i>
-------------	---------------------------

---

**Description**

Query the configuration of the parallel virtual machine.

**Usage**

```
.PVM.config ()
```

**Arguments**

None.

**Details**

.PVM.config returns information about the present virtual machine, similar to that available from the PVM console command conf.

**Value**

Returns the configuration information as a data frame, including each host's pvmd task id, name, architecture and relative speed. One row per host.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**Examples**

```
try (.PVM.config ())
```

---

*.PVM.exit*

*Unregister process from local PVM daemon.*

---

**Description**

Tells the local pvmd that this process is leaving PVM.

**Usage**

```
.PVM.exit()
```

**Arguments**

None.

**Details**

The function *.PVM.exit* tells local pvmd that this process is leaving PVM. This process is not killed but can continue to perform other tasks.

*.PVM.exit* should be called by all PVM process before they stop or exit for good. It *must* be called by processes what were not started with *.PVM.spawn*, such as the master R process.

**Value**

Return 0 when successfully unregistered, -1 when failed.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.mytid](#)

### Examples

```
## Finished with PVM
## Not run: .PVM.exit()
```

---

<code>.PVM.freebuf</code>	<i>Free message buffer</i>
---------------------------	----------------------------

---

### Description

Disposes of a message buffer.

### Usage

```
.PVM.freebuf (bufid)
```

### Arguments

bufid	integer message buffer id
-------	---------------------------

### Details

This function frees the memory associated with the message buffer identified by `bufid`. Message buffers are created by [.PVM.mkbuf](#), [.PVM.initsend](#) and [.PVM.recv](#).

[.PVM.freebuf](#) should be called for a send buffer created by [.PVM.mkbuf](#) after the message has been send and is no longer needed.

Receive buffer typically do not have to be freed unless they have been saved in the courses of using multiple buffers. But [.PVM.freebuf](#) can be used to destroy receive buffers as well. Therefore, messages that arrive but are no longer needed as a result of some other event can be destroyed to save buffer space.

### Value

Returns 0 if successful or -1 if failed.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.initsend](#), [.PVM.mkbuf](#), [.PVM.setrbuf](#), [.PVM.recv](#)

**Examples**

```
## Not run: bufid <- .PVM.mkbuf ("Raw")
## Send the message ...
## Not run: .PVM.freebuf (bufid)
```

---

<code>.PVM.initsend</code>	<i>Initialize send buffer</i>
----------------------------	-------------------------------

---

**Description**

Clears default send buffer and specifies message sending.

**Usage**

```
.PVM.initsend (encoding = c("Default", "Raw", "InPlace"))
```

**Arguments**

`encoding` a character string specifying the next message's encoding scheme. Must be one of "Default" (default), "Raw" or "InPlace".

**Details**

This function clears the send buffer and prepares it for packing a new message.

Possible encoding themes are "Default" (default), "Raw" or "InPlace". If the user knows that the next message will be sent only to a machine that understands the native format, he can use "Raw" to save some encoding costs.

"InPlace" encoding specifies that data be left in place during packing. The message buffer contains only the size and pointers to the items be send. When `.PVM.send` is called, the items are copied directly out of the user's memory. This option decreases the number of times a message is copied, at the expense of requiring that the user not modify the items between the time they are packed and the time they are sent.

`.PVM.encoding` is a mapping of the strings to integers (constants defined by PVM).

**Value**

Returns message buffer id. -1 if there was an error.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

### See Also

[.PVM.mkbuf](#)

### Examples

```
a <- 1:10
## Not run: bufid <- .PVM.initsend ("InPlace")
## Not run: .PVM.pkintvec (a)
## Not run: .PVM.send (tid, msgtag)
```

---

<code>.PVM.kill</code>	<i>Kill pvm process</i>
------------------------	-------------------------

---

### Description

Terminate a specified PVM process.

### Usage

```
.PVM.kill(tids)
```

### Arguments

`tids`                    task ids of the PVM processes to be killed.

### Details

`.PVM.kill` sends a terminate (SIGTERM) signal to the PVM process identified by `tid`. It is not designed to kill the calling process. To kill your self, call [.PVM.exit\(\)](#) followed by `q()`.

### Value

Return number of processes successfully killed.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.exit](#)

### Examples

```
## kill one of spawned children
## Not run: .PVM.kill (children[1])
```

---

*.PVM.mcast**Multicast data*

---

**Description**

Multicast the data in the active message buffer to a set of tasks.

**Usage**

```
.PVM.mcast (tids, msgtag)
```

**Arguments**

tids	vector of the task ids of the tasks to be sent to
msgtag	integer message tag supplied by the user, must be $\geq 0$

**Details**

*.PVM.mcast* multicasts a message stored in the active send buffer to tasks specified in the vector *tids*. The message is not sent to the caller even if its tid is in *tids*. The content of the message can be distinguished by *msgtag*.

The receiving processes can call either [.PVM.recv](#) or [.PVM.nrecv](#) to receive their copy of the multicast. *.PVM.mcast* is asynchronous.

Multicasting is not supported by most multiprocessor vendors. Typically their native calls support only broadcasting to *all* the user's processes on a multiprocessor. Because of this omission, *.PVM.mcast* may not be an efficient communication method on some multiprocessors except in the special case of broadcasting to all PVM processes.

**Value**

Return 0 if successful, -1 if failed.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.send](#)

### Examples

```
## Not run: .PVM.initsend ("Raw")
## Not run: .PVM.pkintvec (1:10)
## Not run: .PVM.mcast (tids, msgtag = 5)
```

---

.PVM.mkbuf	<i>Create message buffer</i>
------------	------------------------------

---

### Description

Creates a new message buffer

### Usage

```
.PVM.mkbuf (encoding = c("Default", "Raw", "InPlace"))
```

### Arguments

encoding            a character string specifying the next message's encoding scheme. Must be one of "Default" (default), "Raw" or "InPlace".

### Details

This function creates a new message buffer and sets its encoding theme (see [.PVM.initsend](#)).

It is useful if the user wishes to manage multiple message buffers and should be used in conjunction with [.PVM.freebuf](#). [.PVM.freebuf](#) should be called for a send buffer after a message has been send and is no longer needed.

Receive buffer are created automatically by [.PVM.recv](#) and [.PVM.nrecv](#) routines and do not have to be freed unless they have been explicitly saved with [.PVM.setrbuf](#).

Typically multiple send and receive buffer are not needed and the user can simply use the [.PVM.initsend](#) to reset the default send buffer.

### Value

Returns message buffer id or -1 if failed..

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.initsend](#), [.PVM.freebuf](#), [.PVM.setrbuf](#)

**Examples**

```
## Not run: bufid <- .PVM.mkbuf ("Raw")
## Send the message ...
## Not run: .PVM.freebuf (bufid)
```

---

*.PVM.mstats**Status of host machines*

---

**Description**

Check the status of hosts computers in the virtual machine.

**Usage**

```
.PVM.mstats (hosts)
```

**Arguments**

hosts                    vector of host names (strings)

**Details**

*.PVM.mstats* returns the status the computers named by *hosts* with respect to running PVM processes. This routine can be used to determine if a particular host has failed and if the virtual machine needs to be reconfigured. [.PVM.notify](#) can also be used to notify the caller that a host has failed.

**Value**

Return machine status. "OK" if host is running, "Not in VM" if host is not in virtual machine, "Not reachable" is host is unreachable (and possibly failed).

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.notify](#)

**Examples**

```
## Not run: .PVM.mstats ("abacus")
```

---

.PVM.mytid

*Task IDs*

---

### Description

Returns the task id(s) of the current, parent and sibling processes.

### Usage

```
.PVM.mytid()  
.PVM.parent ()  
.PVM.siblings ()
```

### Arguments

None.

### Details

.PVM.mytid returns the tid of the calling process and can be called multiple times. The tid is a positive integer created by local pvmd.

This function enrolls this process into PVM on its first call and generate a unique tid if this process was not created by [.PVM.spawn](#). In fact, any PVM system call will enroll a task in PVM if the task is not enrolled before the call.

.PVM.parent returns tid of the process that spawned the calling process. If the calling process was not created with [.PVM.spawn](#), return 0 (no parent).

.PVM.siblings returns the task ids of processes that were spawned together in a single spawn call. The internal list is not updated when sibling tasks exit the virtual machine and should be treated as a snapshot of the parallel program when it was first started. If a task was not started by [.PVM.spawn](#), then [.PVM.siblings](#) will behave identically to [.PVM.mytid](#).

### Value

Returns positive integer task id(s) of the calling process, the parent processor the sibling (started in a single [.PVM.spawn](#) call) processes. Return an negative integer if an error occurred.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.exit](#), [.PVM.spawn](#)

**Examples**

```
## For name(tid) of process and to register with PVM
## Not run:
mytid      <- .PVM.mytid()
myparent   <- .PVM.parent ()
mysiblings <- .PVM.siblings ()

## End(Not run)
```

---

.PVM.notify

*Monitor pvmd*

---

**Description**

Request notification of PVM event such as host failure.

**Usage**

```
.PVM.notify (msgtag, what = c("ExitTask", "DeleteHost", "AddHost"),
            par = 0)
```

**Arguments**

msgtag	message tag to be used in notification
what	a character string specifying the type of event to trigger the notification, must be one of "ExitTask" (default), "DeleteHost", or "AddHost".
par	additional parameter, if what = "AddHost", specify the number of times to notify, if what = "DeleteHost" or "ExitTask", specify a vector of host id or task id.

**Details**

The routine `.PVM.notify` requests PVM to notify the caller on detecting certain events. One or more notify messages (see below) are sent by PVM back to the calling task. The messages have tag `msgtag` supplied to notify.

The notification messages have the following format:

- "ExitTask" One notify message for each task id requested. The message body contains a single integer task id of exited task.
- "DeleteHost" One notify message for each host id requested. The message body contains a single integer host (pvmd) id of exited pvmd.
- "AddHost" `par` notify messages are sent, one each time the local pvmd's host table is updated. The message is a vector of ids of new pvmds and can be unpacked by `.PVM.upkintvec`. The counter of "AddHost" messages yet to be sent is replaced by successive calls to `.PVM.notify`. Specifying `par = -1` turns on "AddHost" messages until a future notify; 0 disables them. The calling task is responsible for receiving messages with the specified tag and taking appropriate action. This feature makes PVM fault tolerant.

**Value**

Status code returned by the routine. -1 if there was an error.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[PVMD](#)

**Examples**

```
## Not run: .PVM.notify (msgtag = 999, what = "Exit", tids)
```

---

.PVM.nrecv	<i>Nonblocking receive</i>
------------	----------------------------

---

**Description**

Checks for nonblocking message.

**Usage**

```
.PVM.nrecv (tid = -1, msgtag = -1)
```

**Arguments**

tid	tid of sending process (-1 matches any tid)
msgtag	integer message tag (>=0) supplied by the user (-1 matches any message tag)

**Details**

.PVM.nrecv checks to see whether a message with lable msgtag has arrived from tid. If a matching message has arrived, it then immediately places the message in a new *active* receive buffer, which also clears the current receive buffer, if any, and returns the buffer id.

If the requested message has not arrived, then .PVM.nrecv immediately returns 0.

A -1 in tid or msgtag matches anything. Later [.PVM.bufinfo](#) can be used to find out the tid of the sending process or msgtag.

.PVM.nrecv is nonblocking in the sense that the routing always returns immediately either with the message or with the information that the message has not arrived at the local pvmd yet. .PVM.nrecv can be called multiple times to check whether a given message has arrived yet.

If .PVM.nrecv returns with the message, the data in the message can be unpacked with the [PVM.unpack](#) functions.

**Value**

Returns the id of the new active receive buffer. -1 indicates an error.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.send](#), [.PVM.recv](#), [.PVM.bufinfo](#), [PVM.unpack](#)

**Examples**

```
## Not run: myparent <- .PVM.parent ()
## Not run: while (1) {
  bufid <- .PVM.nrecv (myparent, 4)
  if (bufid > 0) {
    data <- .PVM.upkintvec ()
    # do something ...
    break
  } else {
    # do something else
  }
}
## End(Not run)
```

---

.PVM.probe

*Probe receive*

---

**Description**

Checks whether message has arrived.

**Usage**

```
.PVM.probe(tid = -1, msgtag = -1)
```

**Arguments**

tid	the specific tid, -1 is every tid
msgtag	tag to use to specific, -1 is all

### Details

Checks to see if a message with label msgtag has arrived from tid. If a matching message has arrived, returns a buffer id which can be used in a [.PVM.bufinfo](#) call to determine information about the message such as its source and length.

A -1 in msgtag or tid matches anything (wildcard).

After the message has arrived, [.PVM.recv](#) must be called before the message can be unpacked into the user's memory using the unpack routines.

### Value

Returns the value of the new active receive buffer id. 0 if the message has not arrived. -1 if there was an error.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.recv](#), [.PVM.bufinfo](#)

### Examples

```
## To check a node (specified by tid) for a message
## Not run: MsgReady <- .PVM.probe (tid, msgtag)
## To see if any node is sending message with tag thisTag
## Not run: AnyMessageWithThisTag <- .PVM.probe (-1, thisTag)
## To see if node tid is sending any message.
## Not run: AnyMessageFromThisTID <- .PVM.probe (tid, -1)
```

---

.PVM.pstats

*Status of PVM processes*

---

### Description

Returns the status of the specified PVM process.

### Usage

`.PVM.pstats (tids)`

### Arguments

tids                    vector of integer task id of destination process

**Details**

The routine `.PVM.pstat` returns the status of the process identified by `tid`. Also note that `.PVM.notify` can be used to notify the caller that a task has failed.

**Value**

Returns the status of the PVM processes ("OK", "Not Running" or "Invalid tid") identified by `tids`.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.notify](#)

**Examples**

```
## Not run: tid <- .PVM.parent ()
## Not run: status <- .PVM.pstats (tid)
```

---

`.PVM.recv`

*Blocking receive*

---

**Description**

Receive a message.

**Usage**

```
.PVM.recv (tid = -1, msgtag = -1)
```

**Arguments**

<code>tid</code>	tid of sending process (-1 matches any tid)
<code>msgtag</code>	integer message tag (>=0) supplied by the user (-1 matches any message tag)

### Details

.PVM.recv blocks the process until a message with label msgtag has arrived from tid. It then places the message in a new *active* receive buffer, which also clears the current receive buffer.

A -1 in tid or msgtag matches anything. Later [.PVM.bufinfo](#) can be used to find out the tid of the sending process or msgtag.

The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both tasks arrive before task 2 does a receive, then a wildcard receive will always return message A.

.PVM.recv is blocking, which means the routing waits until a message matching the user specified tid and msgtag value arrives at the local pvmd.

### Value

Returns the id of the new active receive buffer. -1 indicates an error.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.send](#), [.PVM.nrecv](#), [.PVM.bufinfo](#)

### Examples

```
## Not run: myparent <- .PVM.parent ()
## Not run: bufid <- .PVM.recv (myparent, 4)
## Not run: data <- .PVM.upkintvec ()
```

---

.PVM.send

*Send data*

---

### Description

Sends the data in the active message buffer.

### Usage

.PVM.send (tid, msgtag)

**Arguments**

<code>tid</code>	integer task id of destination process
<code>msgtag</code>	integer message tag ( $\geq 0$ ) supplied by the user

**Details**

Sends a message stored in the active buffer to the PVM process identified by `tid`. `msgtag` is used to label the content of the message.

`.PVM.send` is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication in which computation on the sending processor halts until the matching receive is executed by the receiving processor.

`.PVM.send` first checks to see whether the destination is on the same machine. If so and this host is a multiprocessor, then the vendor's underlying message-passing routines are used to move the data between processes.

**Value**

Returns 0 if successful, -1 if failed..

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.mcast](#), [.PVM.recv](#)

**Examples**

```
a <- 1:10
## Not run:
bufid <- .PVM.initsend ()
.PVM.pkintvec (a)
.PVM.send (tid, msgtag)

## End(Not run)
```

---

.PVM.spawn	<i>Spawn child tasks</i>
------------	--------------------------

---

**Description**

Starts up ntask copies of an executable file task or slave R processes on the virtual machine.

**Usage**

```
.PVM.spawn (task, ntask = 1, flag = "Default", where = "",
            arglist = NULL, verbose = FALSE)
.PVM.spawnR (slave, ntask = 1, flag = "Default", where = "",
            slavedir = "demo", outdir = "/tmp", verbose = FALSE)
.PVM.spawnflags
```

**Arguments**

task	name of an executable file
ntask	number of tasks to spawn
flag	options flag
where	name of a host or architecture
arglist	a character vectpr to be passed to the spawned task.
slave	name of slave R script
slavedir	full path to the slave script
outdir	full path to the directory where the output from slave R process will be
verbose	if true, print out more information which might be helpful in debugging

**Details**

*flag* is a list of strings used to specify options. Allowed values are

"Default"	PVM chooses where to spawn processes
"Host"	spawn on a particular host ( <i>where</i> )
"Arch"	spawn on a particular PVM_ARCH ( <i>where</i> )
"Debug"	starts tasks under a debugger
"Trace"	trace data is generated
"MppFront"	starts tasks on MPP front-end
"HostCompl"	complements host set in <i>where</i>

task is a character string containing the name of the executable file name of the PVM process to be started. It must already reside on the host on which it is to be started and can be find by local pvmd (possibly by specifying ep= option when the host is added, see .PVM.addhosts). The default location PVM looks at is \$PVM\_ROOT/bin/\$PVM\_ARCH.

To spawn a R process, one can use the script 'slaveR.sh' which has to be included in the search

path of pvmd, for example by passing `ep="$R_LIBS/rpvm"` (the default place this script is installed) to `.PVM.addhosts`. Note that here "R\_LIBS" refers to the R library on the *slave* node.

`.PVM.spawnR` can be used to conveniently spawn slave R processes. The paths for the slave script and output file should be given as arguments `toslavedir` and `outdir`.

The name of slave script must be given as argument `slave`. It is crucial that this script is present and can be found on each slave node. The input file path has to be relative to `system.file` (package = "rpvm") on each node, and it has to be the same on all nodes. The default input path is the "demo" subdirectory of `rpvm` installed directory.

The output path can be either relative to the home directory or an absolute pathname (the default is `"/tmp"`). The output file from slave R process will have the same name as the slave script with process id and `.Rout` appended to it.

`.PVM.spawnflags` is a mapping of strings to integer constants defined by PVM.

### Value

A vector containing `tids` of successfully spawned processes (the length could be smaller than the number requested) or -1 if there is a system error.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.mytid](#)

### Examples

```
## Not run:
children <- .PVM.spawnR (slave = "helloR.R")

## End(Not run)
```

---

.PVM.tasks

*Tasks information*

---

### Description

Returns information about the tasks running on the virtual machine

### Usage

`.PVM.tasks` (where = 0)

### Arguments

where integer specifying what tasks to return information about. The options are:

- 0 for all the tasks on the virtual machine.
- host id for all tasks on a given host.
- tid for a specific task.

### Details

`.PVM.tasks` returns information about tasks presently running on the virtual machine. The information returned is the same as that available from the `pvm` console command `ps`.

### Value

A data frame. One row for each task. Names of the columns are,

tid	task id
parent	parent task id
host	host id
status	task running status (see <code>.PVM.pstats</code> ).
name	string, name of the spawned task

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

`.PVM.pstats`, `.PVM.mstats`, `.PVM.config`, `.PVM.tidtohost`

### Examples

```
## Not run:  
.PVM.tasks ()  
  
## End(Not run)
```

---

*.PVM.tidtohost*            *Host id of a task*

---

**Description**

Returns the host id which the specified task is running.

**Usage**

*.PVM.tidtohost* (tid)

**Arguments**

tid                    integer task id specified.

**Value**

Positive integer represents the host id. -1 if failed.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**Examples**

```
## Find out which host this process is running on
## Not run:
myhost <- .PVM.tidtohost (.PVM.mytid())

## End(Not run)
```

---

*.PVM.trecv*                    *Timeout receive*

---

**Description**

Checks for nonblocking message.

**Usage**

*.PVM.trecv* (tid = -1, msgtag = -1, sec = 0, usec = 0)

### Arguments

<code>tid</code>	tid of sending process (-1 matches any tid)
<code>msgtag</code>	integer message tag ( $\geq 0$ ) supplied by the user (-1 matches any message tag)
<code>sec</code>	time to wait in seconds
<code>usec</code>	time to wait in microseconds

### Details

`.PVM.trecv` blocks the process until a message with label `msgtag` has arrived from `tid`. `.PVM.trecv` then places the message in a new active receive buffer, also clearing the current receive buffer. If no matching message arrives within the specified waiting time, `.PVM.trecv` returns without a message.

`sec` and `usec` specify how long `.PVM.trecv` will wait without returning a matching message. With both set to zero, `.PVM.trecv` behaves the same as `.PVM.nrecv`, which is to probe for messages and return immediately even if none are matched. Setting `sec` to -1 makes `.PVM.trecv` behave like `.PVM.trecv`, that is, it will wait indefinitely.

If `.PVM.trecv` is successful, it will return the new active receive buffer identifier. If no message is received, it returns 0. If some error occurs then the return value will be  $< 0$ .

### Value

Returns the id of the new active receive buffer. -1 indicates an error.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.nrecv](#), [.PVM.recv](#)

### Examples

```
## Not run: myparent <- .PVM.parent ()
## Not run: bufid <- .PVM.trecv (myparent, 4, 2)
## Not run: if (bufid > 0) data <- .PVM.upkintvec ()
```

---

init.sprng.master      *Interface to rsprng for initializing SPRNG*

---

### Description

init.sprng.master and init.sprng.slave are used together to initialize the parallel random number generator (PRNG) states in the RPVM master and slaves processes, respectively.

init.sprng.group is used to initialize PRNG states for a RPVM group.

RNGkind ("user") is called at the end of these functions so that the default RNG is substituted with the parallel one.

.SPRNG.INIT.TAG is just a predefined message tag number.

### Usage

```
init.sprng.master (children, seed = 0, kindprng = "default",
                  para = 0)
init.sprng.slave ()
init.sprng.group (group, rootinst = 0, seed = 0,
                  kindprng = "default", para = 0)
.SPRNG.INIT.TAG <- 199
```

### Arguments

children	vector of integer task ids of the slave processes
seed	an integer of random number seed. It is not the starting state of the sequence; rather, it is an encoding of the starting state. The same seed for all the streams. Distinct streams are returned. Only the 31 least significant bits of seed are used in determining the initial starting state of the stream
kindprng	a character string of the disired kind of parallel random number generator
para	additional parameters for the parallel random number generators. If para is 0, default parameters for each PRNG are used. When invalid parameter is given, a warning is issued and the default paramter is used.
group	a character string, the name of the group
rootinst	an integer, the root instance number of the group. The root instance is responsible for getting the seed, rngkind and para parameters from the user

### Value

Return the a two-element character vector of the RNG and normal kinds in use before the call.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

## References

SPRNG: Scalable Parallel Random Number Generator Library Web Page. <http://sprng.cs.fsu.edu/>

## See Also

[init.sprng](#)

## Examples

```
# in master script
## Not run: oldrng <- init.sprng.master (children, seed = 2321)
# in slave script
## Not run: oldrng <- init.sprng.slave ()
# in group processes
## Not run: oldrng <- init.sprng.group ("mygroup", root = 0, seed = 1231)
```

---

PVM.barrier

*Group synchronization*

---

## Description

Blocks the calling process until all processes in a group have called it.

## Usage

```
.PVM.barrier (group, count = .PVM.gsize (group))
```

## Arguments

group	a character string naming the group
count	the number of group members that must call .PVM.barrier before they are all released, usually the total number of members of the specified group.

## Details

.PVM.barrier blocks the calling process until count members of the group have called it. The count argument is required because processes could be joining the given group after other processes have called .PVM.barrier. Thus PVM doesn't know how many group members to wait for at any given instant. Although count can be set less, it is typically the total number of members of the group. So the logical function of the .PVM.barrier call is to provide a group synchronization. During any given barrier call all participating group members must call barrier with the *same* count value. Once a given barrier has been successfully passed, .PVM.barrier can be called again by the same group using the same group name.

## Value

None.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#)

**Examples**

```
gname <- "pvmtest"
## Not run: myinst <- .PVM.joingroup (gname)
# do something here ...
## Not run: .PVM.barrier (gname, 5)
```

---

PVM.bcast

*Broadcasting the data*

---

**Description**

Broadcasts the data in the active message buffer to a group of processes.

**Usage**

```
.PVM.bcast (group, msgtag)
```

**Arguments**

group	a character string naming the group
msgtag	an integer identifying the message

**Details**

`.PVM.bcast` broadcasts a message stored in the active send buffer to all the members of `group`. The broadcast message is not sent back to the sender. Any PVM task can call `.PVM.bcast`, it need not be a member of the group. The content of the message can be distinguished by `msgtag`. At the other end, [.PVM.recv](#) or other receiving functions can be used to receive the message, just as the ones sent by [.PVM.send](#).

`.PVM.bcast` is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processors. This is in contrast to synchronous communication, during which computation on the sending processor halts until a matching receive is executed by all the receiving processors.

`.PVM.bcast` first determines the tids of the group members by checking a group data base. A multicast is performed to these tids. If the group is changed during a broadcast the change will

not be reflected in the broadcast. Multicasting is not supported by most multiprocessor vendors. Typically their native calls only support broadcasting to all the user's processes on a multiprocessor. Because of this omission, `.PVM.bcast` may not be an efficient communication method on some multiprocessors.

**Value**

None.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#)

**Examples**

```
gname <- "pvmtest"
## Not run: .PVM.initsend ()
## Not run: .PVM.pkintvec (1:10)
## Not run: .PVM.bcast(gname, msgtag = 11)
```

---

PVM.buffers

*Manipulating Message Buffers*

---

**Description**

Get or set the id of the active send/receive buffer.

**Usage**

```
.PVM.getsbuf ()
.PVM.getrbuf ()
.PVM.setsbuf (bufid)
.PVM.setrbuf (bufid)
```

**Arguments**

bufid            integer message buffer id

**Details**

The function `.PVM.getsbuf` returns the id of the active send buffer or 0 if no current buffer.

The function `.PVM.getrbuf` returns the id of the active receive buffer or 0 if no current buffer.

The function `.PVM.setrbuf` switches the active receive buffer to `bufid` and saves the previous active receive buffer and returns its id. If `bufid` is set to 0, the present active receive buffer is saved and no active receive buffer exists.

The function `.PVM.setsbuf` switches the active send buffer to `bufid` and saves the previous active send buffer and returns its id. If `bufid` is set to 0, the present active send buffer is saved and no active send buffer exists.

The set functions are required to manage multiple message buffers.

**Value**

The *get* functions return the id of active send/receive buffer. The *set* functions return the id the old send/receive buffer (-1 indicate an error).

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.mkbuf](#), [.PVM.freebuf](#)

**Examples**

```
## Not run: sbufid <- .PVM.getsbuf ()
```

---

PVM.gather

*Gather the data into root*

---

**Description**

A specified member of the group (the root) receives messages from each member of the group and gathers these messages into a single array.

**Usage**

```
.PVM.gather (x, count = length (x), msgtag, group, rootginst = 0)
```

**Arguments**

x	an integer or double vector of length at least count
count	the number of elements to be sent to the root
msgtag	an integer message tag supplied by the user.
group	a character string naming the group
rootginst	an integer instance number of group member who performs the gather of its array to the members of the group.

**Details**

.PVM.gather performs a send of messages from each member of the group to the specified root member of the group. All group members must call .PVM.gather, each sends its vector data of to the root which accumulates these messages into a single vector. The root task is identified by its instance number in the group.

x has to be a vector of storage mode integer or double. The .PVM.gather.default function just calls stop ().

.PVM.gather does not block. If a task calls .PVM.gather and then leaves the group before the root has called .PVM.gather an error may occur.

**Value**

On the root, a vector combining all the x's.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#), [.PVM.reduce](#), [.PVM.scatter](#)

**Examples**

```
gname <- "pvmtest"
## Not run:
if (myinum == 0) {
  result <- .PVM.gather(as.integer (1:100),
                       msgtag = 11, group = gname, root = 0)
}
## End(Not run)
```

---

PVM.getinst

*Instance number identified by group name and task id*

---

### Description

Get the instance numbers identified by a group name and task ids

### Usage

```
.PVM.getinst (group, tids = .PVM.mytid ())
```

### Arguments

group	a character string naming the group
tids	a vector of integer task ids

### Details

.PVM.gettid returns the corresponding instance number of the PVM process identified by the group name group and task ids tids. It can be called by any task whether in the group or not.

### Value

a vector of instance numbers

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[.PVM.joingroup](#), [.PVM.gettid](#)

### Examples

```
gname <- "pvmtest"  
## Not run: myinum <- .PVM.getinst (gname)
```

---

PVM.gettid	<i>Task id identified by group name and instance number</i>
------------	---

---

**Description**

Get the task id identified by a group name and an instance number

**Usage**

```
.PVM.gettid (group, inum = 0:(.PVM.gsize(group) - 1))
```

**Arguments**

group	a character string naming the group
inum	a vector of integer instance numbers

**Details**

.PVM.gettid returns the tid of the PVM process identified by the group name group and the instance number inum.

**Value**

a vector of task ids.

**Note**

inum is given a default value corresponding to all instances in the group. It is not valid when the instances numbers are not continues (when, for example, some processes already left the group).

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#), [.PVM.getinst](#)

**Examples**

```
gname <- "pvmtest"  
leader <- 0  
## Not run: leadtid <- .PVM.gettid (gname, leader)
```

---

PVM.group

*Join or leave a names group*

---

### Description

Enrolls or unenrolls the calling process to a named group.

### Usage

`.PVM.joingroup (group)`

`.PVM.lvgroup (group)`

### Arguments

`group` a character string naming the group

### Details

`.PVM.joingroup` enrolls the calling task in the group named `group` and returns the instance number (`inum`) this task in this group.

Instance numbers start at 0 and count up. When using groups a (`group`, `inum`) pair uniquely identifies a PVM process. If a task leaves a group by calling `.PVM.lvgroup` and later rejoins the same group, the task is not guaranteed to get the same instance number. PVM attempts to reuse old instance numbers, so when a task joins a group it will get the lowest available instance number. A task can be a member of multiple groups simultaneously.

`.PVM.lvgroup` unenrolls the calling task from the group named `group`.

### Value

`.PVM.joingroup` returns the instance number (rank) of the the process in the group.

`.PVM.lvgroup` returns no value.

### Note

If the process fails to join a group with the following message, `libpvm [t40002]: gs_getgclid() failed to start group server: No such file` It means `pvmd` cannot find the executable `pvmsg`, put its path to the `ep=` option of the host file.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

**Examples**

```
gname <- "pvmtest"  
## Not run: myinum <- .PVM.joingroup (gname)  
## Not run: .PVM.lvgroup (gname)
```

---

PVM.gsize	<i>Get the size of the group</i>
-----------	----------------------------------

---

**Description**

Get the number of processes currently in the group

**Usage**

```
.PVM.gsize (group)
```

**Arguments**

group            a character string naming the group

**Value**

Returns the number of processes currently in the group.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#), [.PVM.lvgroup](#)

**Examples**

```
gname <- "pvmtest"  
## Not run: size <- .PVM.gsize(gname)
```

PVM.options

*libpvm Options***Description**

Get or set the value of libpvm options.

**Usage**

PVM.options (what, val)

**Arguments**

what	<p>a character string specifying libpvm options, must be one of the following,</p> <ul style="list-style-type: none"> <li>• "Route" Message routing policy</li> <li>• "DebugMask" Libpvm debug mask</li> <li>• "AutoErr" Auto error reporting</li> <li>• "OutputTid" Stdout destination for children</li> <li>• "OutputCode" Output message tag</li> <li>• "TraceTid" Trace data destination for children</li> <li>• "TraceCode" Trace message tag</li> <li>• "FragSize" Message fragment size</li> <li>• "ResvTids" Allow messages to reserved tags and TIDs</li> <li>• "SelfOutputTid" Stdout destination</li> <li>• "SelfOutputCode" Output message tag</li> <li>• "PvmSelfTraceTid" Trace data destination</li> <li>• "SelfTraceCode" Trace message tag</li> <li>• "ShowTids" pvm_catchout prints task ids with output</li> <li>• "PollType" Message wait policy (shared memory)</li> <li>• "PollTime" Message spinwait duration</li> </ul>
val	integer value of the option

**Details**

.PVM.options sets or queries miscellaneous options in the PVM library. The options are,

1. "Route" Possible values are:
  - 1 "RouteDirect", set up direct task-to-task links (using TCP) for all subsequent communication.
  - 2 "DontRouteDirect", communication through PVM daemon.
  - 3 "AllowDirect" (the default), this setting on task A allows other tasks to set up direct links to A.

2. "DebugMask" When debugging is turned on, PVM will log detailed information about its operations and progress on its stderr stream. The value is the debugging level. Default is 0, not to print any debug information.
3. "AutoErr" When an error results from a libpvm function call and "AutoErr" is set to 1 (the default), an error message is automatically printed on stderr. A setting of 0 disables this. A setting of 2 causes the library to terminate the task by calling exit() after printing the error message. A setting of 3 causes the library to abort after printing the error message.
4. "OutputTid" The stdout destination for children tasks (spawned after the option is set). Everything printed on the standard output of tasks spawned by the calling task is packed into messages and sent to the destination. val is the TID of a PVM task. Setting PvmOutputTid to 0 redirects stdout to the master pvmd, which writes to the log file /tmp/pvml.<uid> The default setting is inherited from the parent task, else is 0.
5. "OutputCode" The message tag for standard output messages. Should only be set when a task has "OutputTid" set to itself.
6. "TraceTid" The trace data message destination for children tasks (spawned after the option is set). Libpvm trace data is sent as messages to the destination. val is the TID of a PVM task. Setting "TraceTid" to 0 discards trace data. The default setting is inherited from the parent task, else is 0.
7. "TraceCode" The message tag for trace data messages. Should only be set when a task has "TraceTid" set to itself.
8. "FragSize" Specifies the message fragment size in bytes. Default value varies with host architecture.
9. "ResvTids" A value of 1 enables the task to send messages with reserved tags and to non-task destinations. The default (0) causes libpvm to generate a "BadParam" error when a reserved identifier is specified.
10. "SelfOutputTid" Sets the stdout destination for the task. Every thing printed on stdout is packed into messages and sent to the destination. Note: this only works for spawned tasks, because the pvmd doesn't get the output from tasks started by other means. The value is the TID of a PVM task. Setting "SelfOutputTid" to 0 redirects stdout to the master pvmd, which writes to the log file /tmp/pvml.<uid>. The default setting is inherited from the parent task, else is 0. Setting either "SelfOutputTid" or "SelfOutputCode" also causes both "OutputTid" and "OutputCode" to take on the values of "SelfOutputTid" and "SelfOutputCode", respectively.
11. "SelfOutputCode" The message tag for standard output messages.
12. "PvmSelfTraceTid" The trace data message destination for the task. Libpvm trace data is sent as messages to the destination. The value is the TID of a PVM task. Setting "SelfTraceTid" to 0 discards trace data. The default setting is inherited from the parent task, else is 0. Setting either "SelfTraceTid" or "SelfTraceCode" also causes both "TraceTid" and "TraceCode" to take on the values of "SelfTraceTid" and "SelfTraceCode", respectively.
13. "SelfTraceCode" The message tag for trace data messages.
14. "ShowTids" If true (nonzero), pvm\_catchout (note: not supported by rpvm) tags each line of output printed by a child task with the task id. Otherwise, output is exactly as printed.
15. "PollType" The message wait policy when using shared-memory message transport. Setting "PollType" to "PollConstant" causes the application to spin on its message queue waiting for a message. Setting "PollType" to "PollSleep" causes the application to poll the message queue for messages "PollTime" times before pending on the semaphore.

16. "PollTime" The poll count for applications checking their message queue before they pend on the semaphore. This option is ignored if "PollType" is set to "PollConstant".

### Value

Returns the current option value. If val is present, set the corresponding option to new value val.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### Examples

```
## Not run: PVM.options (what = "DebugMask")
```

---

PVM.pack

*Packing data*

---

### Description

Pack data into current active send buffer.

### Usage

```
.PVM.pkdouble (data = 0.0, stride = 1)
.PVM.pkint (data = 0, stride = 1)
.PVM.pkstr (data = "")

.PVM.pkintvec (data)
.PVM.pkdblvec (data)
.PVM.pkstrvec (data)

.PVM.pkintmat (data)
.PVM.pkdblmat (data)
.PVM.pkstrmat (data)

.PVM.pkfactor (data)
```

### Arguments

data	data to be packed.
stride	the stride to be used when packing the items. For example, if stride = 2 with .PVM.pkdouble then every other element in the vector data will be packed.

## Details

The first three functions are low-level correspondents of the PVM packing routines for packing double, integer arrays and a single character string. In particular, the number of item packed is not passed and has to be specified when unpacking the data.

The other functions also pack the dimension information therefore there is no need to specify length and the correct dimension will be automatically recovered when corresponding unpack functions are used.

## Value

`.PVM.pkstrvec` returns the maximum length of the strings packed. Others return 0 if succeeded or -1 if failed.

## Note

The PVM C library supports a variety of data types. Since R doesn't have that many types, most are irrelevant.

To passing data between R process, there is in general no need to distinguish between integer and double (except for efficiency consideration). `.PVM.dblvec` and `.PVM.dblmat` are recommended for all numerical data.

`.PVM.pkstr` and `.PVM.upkstr` are mainly for communicating with C functions.

Special values such as NAs, NaNs, Infs, etc., are not supported at this point.

## Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

## References

PVM documentation

## See Also

[PVM.unpack](#)

## Examples

```
## Pack and send a matrix
a <- matrix(1:20, nrow=4)
## Not run: .PVM.pkdblmat(a)
```

PVM.rapply

*Parallel apply*

---

**Description**

Apply a function to the rows of a matrix in parallel, via PVM.

**Usage**

```
PVM.rapply (X, FUN = mean, NTASK = 1)
```

**Arguments**

X	a numeric matrix
FUN	function to be applied on each row
NTASK	number of tasks to use

**Details**

This function requires the function FUN can be correctly evaluated in each spawned R process.

**Value**

A vector with the same values as `apply (X, 1, FUN)`.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**Examples**

```
a <- matrix(rnorm(100), nrow = 10)
## Not run: PVM.rapply (a, sum, 3)
```

---

PVM.reduce

*Reduction*

---

**Description**

Performs a reduction operation over members of the specified group.

**Usage**

```
.PVM.reduce (x, func = "Min", count = length (x), msgtag, group,
             rootginst = 0)
```

**Arguments**

x	an integer or double vector of length at least count
func	a character string specifying the name of the operation (function), must be one of "Min", "Max", "Sum", "Product"
count	the number of elements in x, must be the same among all members of the group
msgtag	an integer message tag supplied by the user.
group	a character string naming the group
rootginst	an integer instance number of group member who performs the reduce of its array to the members of the group.

**Details**

.PVM.reduce performs global operations such as max, min, sum, or a user provided operation on the data provided by the members of a group. All group members call .PVM.reduce with the same size local data array which may contain one or more entries. The root task is identified by its instance number in the

When the .PVM.reduce completes, it returns a vector on the root which will be equal to the specified operation applied element-wise to the data vectors of all the group members.

A broadcast by the root can be used if the other members of the group need the resultant value(s).

func corresponds the four predefined functions in pvm, PvmMin, PvmMax, PvmSum, PvmProduct. It is possible to define user function in C with the following prototype, void func(int \*datatype, void \*x, void \*y, int \*num, int \*info). However, it is not clear how to pass its name from C or how to use user function written in R which limits the usefulness of this function.

**Value**

On the root, a vector of length count.

**Note**

.PVM.reduce does not block, a call to [.PVM.barrier](#) may be necessary. For example, an error may occur if a task calls .PVM.reduce and then leaves the group before the root has completed its call to .PVM.reduce. Similarly, an error may occur if a task joins the group after the root has issued its call to .PVM.reduce. Synchronization of the tasks (such as a call to [.PVM.barrier](#)) was not included within the .PVM.reduce implementation since this overhead is unnecessary in many user codes (which may already synchronize the tasks for other purposes).

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation, <http://www.netlib.org/pvm3/book/pvm-book.html>

**See Also**

[.PVM.joingroup](#), [.PVM.gather](#), [.PVM.scatter](#)

**Examples**

```

gname <- "pvmtest"
## Not run:
if (myinum == 0) {
  result <- .PVM.reduce (as.double (rnorm (10)),
                        "Sum", msgtag = 11, group = gname, root = 0)
}
## End(Not run)

```

---

PVM.scatter

*Scatter a vector across the group*


---

**Description**

Sends to each member of a group a section of a vector from a specified member of the group (the root).

**Usage**

```
.PVM.scatter (x, count, msgtag, group, rootginst = 0)
```

**Arguments**

x	an integer or double vector on the root which are to be distributed to the members of the group. If n is the number of members in the group, then this vector should be of length at least $n \times \text{count}$ . This argument is meaningful only on the root.
count	an integer specifying the number of elements to be sent to each member of the group from the root.
msgtag	an integer message tag supplied by the user.
group	a character string naming the group
rootginst	an integer instance number of group member who performs the scatter of its array to the members of the group.

**Details**

.PVM.scatter performs a scatter of data from the specified root member of the group to each of the members of the group, including itself. All group members must call .PVM.scatter, each receives a portion of the data array from the root in their local result array. It is as if the root node sends to the  $i$ th member of the group count elements from its array data starting at offset  $i \times \text{count}$  from the beginning of the data array. And, it is as if, each member of the group performs a corresponding receive of count values of datatype into its result array. The root task is identified by its instance number in the group.

x has to be a vector of storage mode integer or double. The `.PVM.scatter` function just calls `stop()`.

**Value**

a vector of length count

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**See Also**

[.PVM.joingroup](#), [.PVM.gather](#), [.PVM.reduce](#), [.PVM.bcast](#)

**Examples**

```
gname <- "pvmtest"
## Not run:
myrow <- .PVM.scatter(as.integer(1:100), 10, msgtag = 11, group =
                    gname, root = 0)

## End(Not run)
```

---

PVM.serialize

*Serialize R Objects*

---

**Description**

Serialize to/from the current active send/receive buffer.

**Usage**

```
.PVM.serialize(object, refhook = NULL)
.PVM.unserialize(refhook = NULL)
```

**Arguments**

object            object to serialize.  
refhook           hook function for handling reference objects

**Details**

The function `.PVM.serialize` writes object to the current active send buffer. Sharing of reference objects is preserved within the object but not across separate calls to `.PVM.serialize`. `unserialize` reads an object from connection. connection may also be a string.

The rehook functions can be used to customize handling of non-system reference objects (all external pointers and weak references, and all environments other than name space and package environments and `.GlobalEnv`). The hook function for `.PVM.serialize` should return a character vector for references it wants to handle; otherwise it should return `NULL`. The hook for `.PVM.unserialize` will be called with character vectors supplied to `.PVM.serialize` and should return an appropriate object.

**Value**

`.PVM.serialize` returns `NULL`; `.PVM.unserialize` returns the unserialized object.

**Author(s)**

Luke Tierney <luke@stat.umn.edu>

**Examples**

```
## Pack and send a matrix
## Not run:
BUFTAG<-22
tid <- .PVM.mytid()
a <- matrix (1:20, nrow=4)
.PVM.initsend()
.PVM.serialize(a)
.PVM.send(tid, BUFTAG)
## Receive the matrix
.PVM.recv(tid, BUFTAG)
.PVM.unserialize()

## End(Not run)
```

---

PVM.unpack

*Unpacking data*


---

**Description**

Unpack data from current active receive buffer.

**Usage**

```
.PVM.upkdouble (nitem = 1, stride = 1)
.PVM.upkint (nitem = 1, stride = 1)
.PVM.upkstr (maxlen = 200)
```

```
.PVM.upkintvec ()  
.PVM.upkdblvec ()  
.PVM.upkstrvec ()  
  
.PVM.upkintmat ()  
.PVM.upkdblmat ()  
.PVM.upkstrmat ()  
  
.PVM.upkfactor ()
```

### Arguments

nitem	number of items to unpack.
stride	the stride to be used when unpacking the items. For example, if <code>stride = 2</code> with <code>.PVM.upkdouble</code> then every other element in the buffer will be unpacked.
maxlen	maximum length of the string buffer

### Details

These functions unpack data packed by corresponding `.PVM.pk*` functions. The first two functions `.PVM.upkdbl` and `.PVM.upkint` are lower-level correspondents of the PVM library and require the number of items to unpack as an argument.

`.PVM.upkstr` is used to unpack a string send by C functions or `.PVM.pkstr`. It requires an argument `maxlen` that should be larger than the length of the string passed. For packing and unpacking strings between R processes, `.PVM.pkstrvec` and `.PVM.upkstrvec` can be used (in the case, the maximum length is part of the message).

### Value

Return unpacked data.

### Author(s)

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

### References

PVM documentation

### See Also

[PVM.pack](#)

### Examples

```
## Unpack a matrix  
## Not run: a <- .PVM.upkdblmat ()
```

## Description

Start new pvm daemon, add or delete hosts and stop pvmd.

## Usage

```
.PVM.start.pvmd (hosts = "", block = TRUE)
.PVM.addhosts (hosts)
.PVM.delhosts (hosts)
.PVM.halt ()
```

## Arguments

hosts	a character vector, arguments to pvmd
block	logical, if true, block until startup of all hosts complete

## Details

`.PVM.start.pvmd` starts a *pvmd3* process, the master of a new virtual machine. It returns as soon as the pvmd is started and ready for work. If the `block` parameter is nonzero and a hostfile is passed to the pvmd as a parameter, it returns when all hosts marked to start have been added.

`.PVM.addhosts` takes a vector of host names and add them to the virtual machine. The names should have the same syntax as lines of a pvmd hostfile (see man page for pvmd3): A hostname followed by options of the form `xx=y`.

The status of hosts can be requested by the application using `.PVM.mstats` and `.PVM.config`. If a host fails it will be automatically deleted from the configuration. Using `.PVM.addhosts` a replacement host can be added by the application, however it is the responsibility of the application developer to make his application tolerant of host failure. Another use of this feature would be to add more hosts as they become available, for example on a weekend, or if the application dynamically determines it could use more computational power.

`.PVM.delhosts` deletes the computers pointed to in `hosts` from the existing configuration of computers making up the virtual machine. All PVM processes and the pvmd running on these computers are killed as the computer is deleted.

If a host fails, the PVM system will continue to function and will automatically delete this host from the virtual machine. An application can be notified of a host failure by calling `.PVM.notify`. It is still the responsibility of the application developer to make his application tolerant of host failure.

`.PVM.halt` shuts down the the entire PVM system including remote tasks, remote pvmds, the local tasks (including the calling task) and the local pvmd. Note, when I did this, it also kills the current R session.

**Value**

.PVM.start.pvmd will return the result from .PVM.config.  
.PVM.halt returns TRUE if successful.  
.PVM.addhosts returns a vector of host ids that has been successfully added to the virtual machine.  
.PVM.delhosts returns a vector of status codes for each hosts successfully removed from the virtual machine.

**Author(s)**

Na (Michael) Li <nali@umn.edu> and A.J. Rossini <rossini@u.washington.edu>

**References**

PVM documentation

**Examples**

```
# start a new virtual machine on local machine
## Not run:
.PVM.start.pvmd ()
# add two more hosts to it
.PVM.addhosts ("sparky",
              "thud.cs.utk.edu ep=$R_LIBS/rpvm/")
# do some work ...
# finished with one machine
.PVM.delhosts ("thud.cs.utk.edu")
# do some other work ...
# finished with pvm
.PVM.halt ()

## End(Not run)
```

# Index

## \*Topic **connection**

- .PVM.bufinfo, 2
- .PVM.config, 3
- .PVM.exit, 4
- .PVM.freebuf, 5
- .PVM.initsend, 6
- .PVM.kill, 7
- .PVM.mcast, 8
- .PVM.mkbuf, 9
- .PVM.mstats, 10
- .PVM.mytid, 11
- .PVM.notify, 12
- .PVM.nrecv, 13
- .PVM.probe, 14
- .PVM.pstats, 15
- .PVM.recv, 16
- .PVM.send, 17
- .PVM.spawn, 19
- .PVM.tasks, 20
- .PVM.tidtohost, 22
- .PVM.trecv, 22
- PVM.barrier, 25
- PVM.bcast, 26
- PVM.buffers, 27
- PVM.gather, 28
- PVM.getinst, 30
- PVM.gettid, 31
- PVM.group, 32
- PVM.gsize, 33
- PVM.options, 34
- PVM.pack, 36
- PVM.rapply, 38
- PVM.reduce, 38
- PVM.scatter, 40
- PVM.serialize, 41
- PVM.unpack, 42
- PVMD, 44

## \*Topic **distribution**

- init.sprng.master, 24

## \*Topic **interface**

- .PVM.bufinfo, 2
- .PVM.config, 3
- .PVM.exit, 4
- .PVM.freebuf, 5
- .PVM.initsend, 6
- .PVM.kill, 7
- .PVM.mcast, 8
- .PVM.mkbuf, 9
- .PVM.mstats, 10
- .PVM.mytid, 11
- .PVM.notify, 12
- .PVM.nrecv, 13
- .PVM.probe, 14
- .PVM.pstats, 15
- .PVM.recv, 16
- .PVM.send, 17
- .PVM.spawn, 19
- .PVM.tasks, 20
- .PVM.tidtohost, 22
- .PVM.trecv, 22
- init.sprng.master, 24
- PVM.barrier, 25
- PVM.bcast, 26
- PVM.buffers, 27
- PVM.gather, 28
- PVM.getinst, 30
- PVM.gettid, 31
- PVM.group, 32
- PVM.gsize, 33
- PVM.options, 34
- PVM.pack, 36
- PVM.rapply, 38
- PVM.reduce, 38
- PVM.scatter, 40
- PVM.serialize, 41
- PVM.unpack, 42
- PVMD, 44

## \*Topic **utilities**

- .PVM.bufinfo, 2
- .PVM.config, 3
- .PVM.exit, 4
- .PVM.freebuf, 5
- .PVM.initsend, 6
- .PVM.kill, 7
- .PVM.mcast, 8
- .PVM.mkbuf, 9
- .PVM.mstats, 10
- .PVM.mytid, 11
- .PVM.notify, 12
- .PVM.nrecv, 13
- .PVM.probe, 14
- .PVM.pstats, 15
- .PVM.recv, 16
- .PVM.send, 17
- .PVM.spawn, 19
- .PVM.tasks, 20
- .PVM.tidtohost, 22
- .PVM.trecv, 22
- PVM.barrier, 25
- PVM.bcast, 26
- PVM.buffer, 27
- PVM.gather, 28
- PVM.getinst, 30
- PVM.gettid, 31
- PVM.group, 32
- PVM.gsize, 33
- PVM.options, 34
- PVM.pack, 36
- PVM.rapply, 38
- PVM.reduce, 38
- PVM.scatter, 40
- PVM.serialize, 41
- PVM.unpack, 42
- PVMD, 44
- .PVM.addhosts, 19, 44
- .PVM.addhosts (PVMD), 44
- .PVM.barrier, 39
- .PVM.barrier (PVM.barrier), 25
- .PVM.bcast, 41
- .PVM.bcast (PVM.bcast), 26
- .PVM.bufinfo, 2, 13–15, 17
- .PVM.config, 3, 21, 44
- .PVM.delhosts (PVMD), 44
- .PVM.encoding (.PVM.initsend), 6
- .PVM.exit, 4, 7, 11
- .PVM.freebuf, 5, 5, 9, 28
- .PVM.gather, 40, 41
- .PVM.gather (PVM.gather), 28
- .PVM.getinst, 31
- .PVM.getinst (PVM.getinst), 30
- .PVM.getrbuf (PVM.buffer), 27
- .PVM.getsbuf (PVM.buffer), 27
- .PVM.gettid, 30
- .PVM.gettid (PVM.gettid), 31
- .PVM.gsize (PVM.gsize), 33
- .PVM.halt (PVMD), 44
- .PVM.initsend, 5, 6, 9
- .PVM.joingroup, 26, 27, 29–31, 33, 40, 41
- .PVM.joingroup (PVM.group), 32
- .PVM.kill, 7
- .PVM.lvgroup, 33
- .PVM.lvgroup (PVM.group), 32
- .PVM.mcast, 8, 18
- .PVM.mkbuf, 5, 7, 9, 28
- .PVM.mstats, 10, 21, 44
- .PVM.mytid, 4, 11, 20
- .PVM.notify, 10, 12, 16, 44
- .PVM.nrecv, 3, 8, 9, 13, 17, 23
- .PVM.parent (.PVM.mytid), 11
- .PVM.pkdblmat (PVM.pack), 36
- .PVM.pkdblvec (PVM.pack), 36
- .PVM.pkdouble (PVM.pack), 36
- .PVM.pkfactor (PVM.pack), 36
- .PVM.pkint (PVM.pack), 36
- .PVM.pkintmat (PVM.pack), 36
- .PVM.pkintvec (PVM.pack), 36
- .PVM.pkstr, 43
- .PVM.pkstr (PVM.pack), 36
- .PVM.pkstrmat (PVM.pack), 36
- .PVM.pkstrvec, 43
- .PVM.pkstrvec (PVM.pack), 36
- .PVM.probe, 14
- .PVM.pstats, 15, 21
- .PVM.recv, 3, 5, 8, 9, 14, 15, 16, 18, 23, 26
- .PVM.reduce, 29, 41
- .PVM.reduce (PVM.reduce), 38
- .PVM.scatter, 29, 40
- .PVM.scatter (PVM.scatter), 40
- .PVM.send, 3, 6, 8, 14, 17, 17, 26
- .PVM.serialize (PVM.serialize), 41
- .PVM.setrbuf, 5, 9
- .PVM.setrbuf (PVM.buffer), 27
- .PVM.setsbuf (PVM.buffer), 27
- .PVM.siblings (.PVM.mytid), 11

- .PVM.spawn, [11](#), [19](#)
- .PVM.spawnR (.PVM.spawn), [19](#)
- .PVM.spawnflags (.PVM.spawn), [19](#)
- .PVM.start.pvmd (PVMD), [44](#)
- .PVM.tasks, [20](#)
- .PVM.tidtohost, [21](#), [22](#)
- .PVM.trecv, [22](#), [23](#)
- .PVM.unserialize (PVM.serialize), [41](#)
- .PVM.upkdblmat (PVM.unpack), [42](#)
- .PVM.upkdblvec (PVM.unpack), [42](#)
- .PVM.upkdouble (PVM.unpack), [42](#)
- .PVM.upkfactor (PVM.unpack), [42](#)
- .PVM.upkint (PVM.unpack), [42](#)
- .PVM.upkintmat (PVM.unpack), [42](#)
- .PVM.upkintvec, [12](#)
- .PVM.upkintvec (PVM.unpack), [42](#)
- .PVM.upkstr, [37](#)
- .PVM.upkstr (PVM.unpack), [42](#)
- .PVM.upkstrmat (PVM.unpack), [42](#)
- .PVM.upkstrvec (PVM.unpack), [42](#)
- .SPRNG.INIT.TAG (init.sprng.master), [24](#)

init.sprng, [25](#)  
init.sprng.group (init.sprng.master), [24](#)  
init.sprng.master, [24](#)  
init.sprng.slave (init.sprng.master), [24](#)

PVM.barrier, [25](#)  
PVM.bcast, [26](#)  
PVM.buffers, [27](#)  
PVM.gather, [28](#)  
PVM.getinst, [30](#)  
PVM.gettid, [31](#)  
PVM.group, [32](#)  
PVM.gsize, [33](#)  
PVM.options, [34](#)  
PVM.pack, [36](#), [43](#)  
PVM.rapply, [38](#)  
PVM.reduce, [38](#)  
PVM.scatter, [40](#)  
PVM.serialize, [41](#)  
PVM.unpack, [13](#), [14](#), [37](#), [42](#)  
PVMD, [13](#), [44](#)