Package ‘Rmpi’

January 20, 2025

Version 0.7-3.3

Date 2025-01-13

Title Interface (Wrapper) to MPI (Message-Passing Interface)

Depends R (>= 2.15.1)

Imports parallel

Description An interface (wrapper) to MPI. It also
provides interactive R manager and worker environment.

License GPL (>= 2)

URL https://fisher.stats.uwo.ca/faculty/yu/Rmpi/

Maintainer Hao Yu <hyu@stats.uwo.ca>

Author Hao Yu [aut, cre]

NeedsCompilation yes

Repository CRAN

Date/Publication 2025-01-13 16:50:02 UTC

Contents

`lamhosts` *Hosts Information*

Description

`lamhosts` finds the host name associated with its node number. Can be used by `mpi.spawn.Rslaves` to spawn R slaves on selected hosts. This is a LAM-MPI specific function.

`mpi.is.master` checks if it is running on master or slaves.

`mpi.hostinfo` finds an individual host information including rank and size in a comm.

`slave.hostinfo` is executed only by master and find all master and slaves host information in a comm.

Usage

```
lamhosts()
mpi.is.master()
mpi.hostinfo(comm = 1)
slave.hostinfo(comm = 1, short=TRUE)
```

Arguments

`comm`	a communicator number
`short`	if true, a short form is printed

Value

`lamhosts` returns CPUs nodes numbers with their host names.

`mpi.is.master` returns TRUE if it is on master and FALSE otherwise.

`mpi.hostinfo` sends to stdio a host name, rank, size and comm.

`slave.hostname` sends to stdio a list of host, rank, size, and comm information for all master and slaves. With short=TRUE and 8 slaves or more, the first 3 and last 2 slaves are shown.

Author(s)

Hao Yu

See Also

`mpi.spawn.Rslaves`

`mpi.abort` *MPI_Abort API*

Description

`mpi.abort` makes a “best attempt" to abort all tasks in a comm.

Usage

```
mpi.abort(comm = 1)
```

Arguments

`comm`	a communicator number

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.finalize`

`mpi.any.source` *MPI Constants*

Description

Find MPI constants: MPI_ANY_SOURCE, MPI_ANY_TAG, or MPI_PROC_NULL

Usage

```
mpi.any.source()
mpi.any.tag()
mpi.proc.null()
```

Arguments

None

Details

These constants are mainly used by `mpi.send`, `mpi.recv`, and `mpi.probe`. Different implementation of MPI may use different integers for MPI_ANY_SOURCE, MPI_ANY_TAG, and MPI_PROC_NULL. Hence one should use these functions instead real integers for MPI communications.

Value

Each function returns an integer value.

References

https://www.open-mpi.org/

See Also

`mpi.send`, `mpi.recv`.

`mpi.apply` *Scatter an array to slaves and then apply a FUN*

Description

An array (length <= total number of slaves) is scattered to slaves so that the first slave calls `FUN` with arguments `x[[1]]` and `...`, the second one calls with arguments `x[[2]]` and `...`, and so on. `mpi.iapply` is a nonblocking version of `mpi.apply` so that it will not consume CPU on master node.

Usage

```
mpi.apply(X, FUN, ..., comm=1)
mpi.iapply(X, FUN, ..., comm=1, sleep=0.01)
```

Arguments

`X`	an array
`FUN`	a function
`...`	optional arguments to `FUN`
`comm`	a communicator number
`sleep`	a sleep interval on master node (in sec)

Value

A list of the results is returned. Its length is the same as that of `x`. In case the call `FUN` with arguments `x[[i]]` and `...` fails on ith slave, corresponding error message will be returned in the returning list.

Author(s)

Hao Yu

Examples

```
#Assume that there are at least 5 slaves running
#Otherwise run mpi.spawn.Rslaves(nslaves=5)
#x=c(10,20)
#mpi.apply(x,runif)
#meanx=1:5
#mpi.apply(meanx,rnorm,n=2,sd=4)
```

mpi.applyLB *(Load balancing) parallel apply*

Description

(Load balancing) parallelapply and related functions.

Usage

```
mpi.applyLB(X, FUN, ..., apply.seq=NULL, comm=1)
mpi.parApply(X, MARGIN, FUN, ..., job.num = mpi.comm.size(comm)-1,
                    apply.seq=NULL, comm=1)
mpi.parLapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
comm=1)
mpi.parSapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
simplify=TRUE, USE.NAMES = TRUE, comm=1)
mpi.parRapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
comm=1)
mpi.parCapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
comm=1)
mpi.parReplicate(n, expr, job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
simplify = "array", comm=1)
mpi.parMM (A, B, job.num=mpi.comm.size(comm)-1, comm=1)
```

Arguments

`X`	an array or matrix.
`MARGIN`	vector specifying the dimensions to use.
`FUN`	a function.
`simplify`	logical or character string; should the result be simplified to a vector, matrix or higher dimensional array if possible?
`USE.NAMES`	logical; if `TRUE` and if `X` is character, use `X` as `names` for the result unless it had names already.
`n`	number of replications.
`A`	a matrix
`B`	a matrix
`expr`	expression to evaluate repeatedly.
`job.num`	Total job numbers. If job numbers is bigger than total slave numbers (default value), a load balancing approach is used.
`apply.seq`	if reproducing the same computation (simulation) is desirable, set it to the integer vector .mpi.applyLB generated in previous computation (simulation).
`...`	optional arguments to `FUN`
`comm`	a communicator number

Details

Unless length of `X` is no more than total slave numbers (slave.num) and in this case `mpi.applyLB` is the same as `mpi.apply`, `mpi.applyLB` sends a next job to a slave who just delivered a finished job. The sequence of slaves who deliver results to master are saved into `.mpi.applyLB`. It keeps track which part of results done by which slaves. `.mpi.applyLB` can be used to reproduce the same simulation result if the same seed is used and the argument `apply.seq` is equal to `.mpi.applyLB`.

With the default value of argument `job.num` which is slave.num, `mpi.parApply`, `mpi.parLapply`, `mpi.parSapply`, `mpi.parRapply`, `mpi.parCapply`, `mpi.parSapply`, and `mpi.parMM` are clones of **snow**'s parApply, parLappy, parSapply, parRapply, parCapply, parSapply, and parMM, respectively. When `job.num` is bigger than slave.num, a load balancing approach is used.

Warning

When using the argument `apply.seq` with `.mpi.applyLB`, be sure all settings are the same as before, i.e., the same data, job.num, slave.num, and seed. Otherwise a deadlock could occur. Notice that `apply.seq` is useful only if `job.num` is bigger than slave.num.

See Also

`mpi.apply`

Examples

```
#Assume that there are some slaves running

#mpi.applyLB
#x=1:7
#mpi.applyLB(x,rnorm,mean=2,sd=4)

#get the same simulation
#mpi.remote.exec(set.seed(111))
#mpi.applyLB(x,rnorm,mean=2,sd=4)
#mpi.remote.exec(set.seed(111))
#mpi.applyLB(x,rnorm,mean=2,sd=4,apply.seq=.mpi.applyLB)

#mpi.parApply
#x=1:24
#dim(x)=c(2,3,4)
#mpi.parApply(x, MARGIN=c(1,2), FUN=mean,job.num = 5)

#mpi.parLapply
#mdat <- matrix(c(1,2,3, 7,8,9), nrow = 2, ncol=3, byrow=TRUE,
#                    dimnames = list(c("R.1", "R.2"), c("C.1", "C.2", "C.3")))
#mpi.parLapply(mdat, rnorm)

#mpi.parSapply
#mpi.parSapply(mdat, rnorm)

#mpi.parMM
#A=matrix(1:1000^2,ncol=1000)
#mpi.parMM(A,A)
```

`mpi.barrier` *MPI_Barrier API*

Description

`mpi.barrier` blocks the caller until all members have called it.

Usage

```
mpi.barrier(comm = 1)
```

Arguments

`comm`	a communicator number

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

`mpi.bcast` *MPI_Bcast API*

Description

`mpi.bcast` is a collective call among all members in a comm. It broadcasts a message from the specified rank to all members.

Usage

```
mpi.bcast(x, type, rank = 0, comm = 1, buffunit=100)
```

Arguments

`x`	data to be sent or received. Must be the same type among all members.
`type`	1 for integer, 2 for double, and 3 for character. Others are not supported.
`rank`	the sender.
`comm`	a communicator number.
`buffunit`	a buffer unit number.

Details

`mpi.bcast` is a blocking call among all members in a comm, i.e, all members have to wait until everyone calls it. All members have to prepare the same type of messages (buffers). Hence it is relatively difficult to use in R environment since the receivers may not know what types of data to receive, not mention the length of data. Users should use various extensions of `mpi.bcast` in R. They are `mpi.bcast.Robj`, `mpi.bcast.cmd`, and `mpi.bcast.Robj2slave`.

When type=5, MPI continuous datatype (double) is defined with unit given by `buffunit`. It is used to transfer huge data where a double vector or matrix is divided into many chunks with unit `buffunit`. Total ceiling(length(obj)/buffunit) units are transferred. Due to MPI specification, both `buffunit` and total units transferred cannot be over 2^31-1. Notice that the last chunk may not have full length of data due to rounding. Special care is needed.

Value

`mpi.bcast` returns the message broadcasted by the sender (specified by the rank).

References

`https://www.open-mpi.org/`

See Also

`mpi.bcast.Robj`, `mpi.bcast.cmd`, `mpi.bcast.Robj2slave`.

`mpi.bcast.cmd` *Extension of MPI_Bcast API*

Description

`mpi.bcast.cmd` is an extension of `mpi.bcast`. It is mainly used to transmit a command from master to all R slaves spawned by using slavedaemon.R script.

Usage

```
mpi.bcast.cmd(cmd=NULL, ..., rank = 0, comm = 1, nonblock=FALSE, sleep=0.1)
```

Arguments

`cmd`	a command to be sent from master.
`...`	used as arguments to cmd (function command) for passing their (master) values to R slaves, i.e., if ‘myfun(x)’ will be executed on R slaves with ‘x’ as master variable, use mpi.bcast.cmd(cmd=myfun, x=x).
`rank`	the sender
`comm`	a communicator number
`nonblock`	logical. If TRUE, a nonblock procedure is used on all receivers so that they will consume none or little CPUs while waiting.
`sleep`	a sleep interval, used when nonblock=TRUE. Smaller sleep is, more response receivers are, more CPUs consume

Details

`mpi.bcast.cmd` is a collective call. This means all members in a communicator must execute it at the same time. If slaves are spawned (created) by using slavedaemon.R (Rprofile script), then they are running `mpi.bcast.cmd` in infinite loop (idle state). Hence master can execute `mpi.bcast.cmd` alone to start computation. On the master, `cmd` and `...` are put together as a list which is then broadcasted (after serialization) to all slaves (using for loop with mpi.send and mpi.recv pair). All slaves will return an expression which will be evaluated by either slavedaemon.R, or by whatever an R script based on slavedaemon.R.

If nonblock=TRUE, then on receiving side, a nonblock procedure is used to check if there is a message. If not, it will sleep for the specied amount and repeat itself.

Please use `mpi.remote.exec` if you want the executed results returned from R slaves.

Value

`mpi.bcast.cmd` returns no value for the sender and an expression of the transmitted command for others.

Warning

Be caution to use `mpi.bcast.cmd` alone by master in the middle of comptuation. Only all slaves in idle states (waiting instructions from master) can it be used. Othewise it may result miscommunication with other MPI calls.

Author(s)

Hao Yu

See Also

`mpi.remote.exec`

`mpi.bcast.Robj` *Extensions of MPI_Bcast API*

Description

`mpi.bcast.Robj` and `mpi.bcast.Robj2slave` are used to move a general R object around among master and all slaves.

Usage

```
mpi.bcast.Robj(obj = NULL, rank = 0, comm = 1)
mpi.bcast.Robj2slave(obj, comm = 1, all = FALSE)
mpi.bcast.Rfun2slave(comm = 1)
mpi.bcast.data2slave(obj, comm = 1, buffunit = 100)
```

Arguments

`obj`	an R object to be transmitted from the sender
`rank`	the sender.
`comm`	a communicator number.
`all`	a logical. If TRUE, all R objects on master are transmitted to slaves.
`buffunit`	a buffer unit number.

Details

`mpi.bcast.Robj` is an extension of `mpi.bcast` for moving a general R object around from a sender to everyone. `mpi.bcast.Robj2slave` does an R object transmission from master to all slaves unless all=TRUE in which case, all master's objects with the global enviroment are transmitted to all slavers.

`mpi.bcast.data2slave` transfers data (a double vector or a matrix) natively without (un)serilization. It should be used with a huge vector or matrix. It results less memory usage and faster transmission. Notice that data with missing values (NA) are allowed.

Value

`mpi.bcast.Robj` returns no value for the sender and the transmitted one for others. `mpi.bcast.Robj2slave` returns no value for the master and the transmitted R object along its name on slaves. `mpi.bcast.Rfun2slave` transmits all master's functions to slaves and returns no value. `mpi.bcast.data2slave` transmits a double vector or a matrix to slaves and returns no value.

Author(s)

Hao Yu

See Also

`mpi.send.Robj`, `mpi.recv.Robj`,

`mpi.cart.coords` *MPI_Cart_coords*

Description

`mpi.cart.coords` translates a rank to its Cartesian topology coordinate.

Usage

```
mpi.cart.coords(comm=3, rank, maxdims)
```

Arguments

`comm`	Communicator with Cartesian structure
`rank`	rank of a process within group
`maxdims`	length of vector coord in the calling program

Details

This function is the rank-to-coordinates translator. It is the inverse map of `mpi.cart.rank`. maxdims is at least as big as ndims as returned by `mpi.cartdim.get`.

Value

`mpi.cart.coords` returns an integer array containing the Cartesian coordinates of specified process.

Author(s)

Alek Hunchak and Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.cart.rank`

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
#mpi.cart.coords(3,4,2)
```

`mpi.cart.create` *MPI_Cart_create*

Description

`mpi.cart.create` creates a Cartesian structure of arbitrary dimension.

Usage

```
mpi.cart.create(commold=1, dims, periods, reorder=FALSE, commcart=3)
```

Arguments

`commold`	Input communicator
`dims`	Integery array of size ndims specifying the number of processes in each dimension
`periods`	Logical array of size ndims specifying whether the grid is periodic or not in each dimension
`reorder`	ranks may be reordered or not
`commcart`	The new communicator to which the Cartesian topology information is attached

Details

If reorder = false, then the rank of each process in the new group is the same as its rank in the old group. If the total size of the Cartesian grid is smaller than the size of the group of commold, then some processes are returned mpi.comm.null. The call is erroneous if it specifies a grid that is larger than the group size.

Value

`mpi.cart.create` returns 1 if success and 0 otherwise.

Author(s)

Alek Hunchak and Hao Yu

References

https://www.open-mpi.org/

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
```

`mpi.cart.get` *MPI_Cart_get*

Description

`mpi.cart.get` provides the user with information on the Cartesian topology associated with a comm.

Usage

```
 mpi.cart.get(comm=3, maxdims)
```

Arguments

`comm`	Communicator with Cartesian structure
`maxdims`	length of vectors dims, periods, and coords in the calling program

Details

The coords are as given for the rank of the calling process as shown.

Value

`mpi.cart.get` returns a vector containing information on the Cartesian topology associated with comm. maxdims must be at least ndims as returned by `mpi.cartdim.get`.

Author(s)

Alek Hunchak and Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.cart.create`,`mpi.cartdim.get`

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
#mpi.remote.exec(mpi.cart.get(3,2))
```

`mpi.cart.rank` *MPI_Cart_rank*

Description

`mpi.cart.rank` translates a Cartesian topology coordinate to its rank.

Usage

```
 mpi.cart.rank(comm=3, coords)
```

Arguments

`comm`	Communicator with Cartesian structure
`coords`	Specifies the Cartesian coordinates of a process

Details

For a process group with a Cartesian topology, this function translates the logical process coordinates to process ranks as they are used by the point-to-point routines. It is the inverse map of `mpi.cart.coords`.

Value

`mpi.cart.rank` returns the rank of the specified process.

Author(s)

Alek Hunchak and Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.cart.coords`

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
#mpi.cart.rank(3,c(1,0))
```

`mpi.cart.shift` *MPI_Cart_shift*

Description

`mpi.cart.shift` shifts the Cartesian topology in both manners, displacement and direction.

Usage

```
mpi.cart.shift(comm=3, direction, disp)
```

Arguments

`comm`	Communicator with Cartesian structure
`direction`	Coordinate dimension of the shift
`disp`	displacement (>0 for upwards or left shift, <0 for downwards or right shift)

Details

`mpi.cart.shift` provides neighbor ranks from given direction and displacement. The direction argument indicates the dimension of the shift. direction=1 means the first dim, direction=2 means the second dim, etc. disp=1 or -1 provides immediate neighbor ranks and disp=2 or -2 provides neighbor's neighbor ranks. Negative ranks mean out of boundary. They correspond to `mpi.proc.null`.

Value

`mpi.cart.shift` returns a vector containing information regarding the rank of the source process and rank of the destination process.

Author(s)

Alek Hunchak and Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.cart.create`,`mpi.proc.null`

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
#mpi.remote.exec(mpi.cart.shift(3,2,1))#get neighbor ranks
#mpi.remote.exec(mpi.cart.shift(3,1,1))
```

`mpi.cartdim.get` *MPI_Cartdim_get*

Description

`mpi.cartdim.get` gets dim information about a Cartesian topology.

Usage

```
mpi.cartdim.get(comm=3)
```

Arguments

`comm`	Communicator with Cartesian structure

Details

Can be used to provide other functions with the correct size of arrays.

Value

`mpi.cartdim.get` returns the number of dimensions of the Cartesian structure

Author(s)

Alek Hunchak and Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.cart.get`

Examples

```
#Need at least 9 slaves
#mpi.bcast.cmd(mpi.cart.create(1,c(3,3),c(F,T)))
#mpi.cart.create(1,c(3,3),c(F,T))
#mpi.cartdim.get(comm=3)
```

`mpi.comm.disconnect` *MPI_Comm_disconnect API*

Description

`mpi.comm.disconnect` disconnects itself from a communicator and then deallocates the communicator so it points to MPI_COMM_NULL.

Usage

```
mpi.comm.disconnect(comm=1)
```

Arguments

`comm`	a communicator number

Details

When members associated with a communicator finish jobs or exit, they have to call `mpi.comm.disconnect` to release resource if the communicator was created from an intercommunicator by `mpi.intercomm.merge`. If `mpi.comm.free` is used instead, `mpi.finalize` called by slaves may cause undefined impacts on master who wishes to stay.

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

mpi.comm.free

---

mpi.comm.free *MPI_Comm_free API*

Description

mpi.comm.free deallocates a communicator so it points to MPI_COMM_NULL.

Usage

```
mpi.comm.free(comm=1)
```

Arguments

comm	a communicator number

Details

When members associated with a communicator finish jobs or exit, they have to call mpi.comm.free to release resource so mpi.comm.size will return 0. If the comm was created from an intercommunicator by mpi.intercomm.merge, use mpi.comm.disconnect instead.

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.comm.disconnect`

`mpi.comm.get.parent` *MPI_Comm_get_parent, MPI_Comm_remote_size, MPI_Comm_test_inter APIs*

Description

`mpi.comm.get.parent` is mainly used by slaves to find the intercommunicator or the parent who spawns them. The intercommunicator is saved in the specified comm number.

`mpi.comm.remote.size` is mainly used by master to find the total number of slaves spawned.

`mpi.comm.test.inter` tests if a comm is an intercomm or not.

Usage

```
mpi.comm.get.parent(comm = 2)
mpi.comm.remote.size(comm = 2)
mpi.comm.test.inter(comm = 2)
```

Arguments

`comm`	an intercommunicator number.

Value

`mpi.comm.get.parent` and `mpi.comm.test.inter` return 1 if success and 0 otherwise.

`mpi.comm.remote.size` returns the total number of members in the remote group in an intercomm.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.intercomm.merge`

`mpi.comm.set.errhandler` *MPI_Comm_set_errhandler API*

Description

`mpi.comm.set.errhandler` sets a communicator to MPI_ERRORS_RETURN instead of MPI_ERRORS_ARE_FATAL (default) which crashes R on any type of MPI errors. Almost all MPI API calls return errcodes which can map to specific MPI error messages. All MPI related error messages come from predefined MPI_Error_string.

Usage

```
mpi.comm.set.errhandler(comm = 1)
```

Arguments

`comm`	a communicator number

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

`mpi.comm.size` *MPI_Comm_c2f, MPI_Comm_dup, MPI_Comm_rank, and MPI_Comm_size APIs*

Description

`mpi.comm.c2f` converts the comm (a C communicator) and returns an integer that can be used as the communicator in external FORTRAN code. `mpi.comm.dup` duplicates (copies) a comm to a new comm. `mpi.comm.rank` returns its rank in a comm. `mpi.comm.size` returns the total number of members in a comm.

Usage

```
mpi.comm.c2f(comm=1)
mpi.comm.dup(comm, newcomm)
mpi.comm.rank(comm = 1)
mpi.comm.size(comm = 1)
```

Arguments

`comm`	a communicator number
`newcomm`	a new communicator number

Author(s)

Hao Yu

References

https://www.open-mpi.org/

Examples

```
#Assume that there are some slaves running
#mpi.comm.size(comm=1)
#mpi.comm.size(comm=0)

#mpi.remote.exec(mpi.comm.rank(comm=1))
#mpi.remote.exec(mpi.comm.rank(comm=0))

#mpi.remote.exec(mpi.comm.size(comm=1))
#mpi.remote.exec(mpi.comm.size(comm=0))

#mpi.bcast.cmd(mpi.comm.dup(comm=1,newcomm=5))
#mpi.comm.dup(comm=1,newcomm=5)
```

`mpi.comm.spawn` *MPI_Comm_spawn API*

Description

`mpi.comm.spawn` tries to start `nslaves` identical copies of `slaves`, establishing communication with them and returning an intercommunicator. The spawned slaves are referred to as children, and the process that spawned them is called the parent (master). The children have their own MPI_COMM_WORLD represented by comm 0. To make communication possible among master and slaves, all slaves should use `mpi.comm.get.parent` to find their parent and use `mpi.intercomm.merge` to merger an intercomm to a comm.

Usage

```
 mpi.comm.spawn(slave, slavearg = character(0),
               nslaves = mpi.universe.size(), info = 0,
               root = 0, intercomm = 2, quiet = FALSE)
```

Arguments

`slave`	a file name to an executable program.
`slavearg`	an argument list (a char vector) to slave.
`nslaves`	number of slaves to be spawned.
`info`	an info number.
`root`	the root member who spawns slaves.
`intercomm`	an intercomm number.
`quiet`	a logical. If TRUE, do not print anything unless an error occurs.

Value

Unless `quiet = TRUE`, a message is printed to indicate how many slaves are successfully spawned and how many failed.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.comm.get.parent`, `mpi.intercomm.merge`.

`mpi.dims.create` *MPI_Dims_create*

Description

`mpi.dims.create` Create a Cartesian dimension used by `mpi.cart.create`.

Usage

```
mpi.dims.create(nnodes, ndims, dims=integer(ndims))
```

Arguments

`nnodes`	Number of nodes in a cluster
`ndims`	Number of dimension in a Cartesian topology
`dims`	Initial dimension numbers

Details

The entries in the return value are set to describe a Cartesian grid with `ndims` dimensions and a total of `nnodes` nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The return value can be constrained by specifying positive number(s) in `dims`. Only those 0 values in `dims` are modified by `mpi.dims.create`.

Value

`mpi.dims.create` returns the dimension vector used by that in `mpi.cart.create`.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.cart.create`

Examples

```
#What is the dim numbers of 2 dim Cartersian topology under a grid of 36 nodes
#mpi.dims.create(36,2) #return c(6,6)

#Constrained dim numbers
#mpi.dims.create(12,2,c(0,4)) #return c(9,4)
```

`mpi.exit` *Exit MPI Environment*

Description

`mpi.exit` terminates MPI execution environment and detaches the library Rmpi. After that, you can still work on R.

`mpi.quit` terminates MPI execution environment and quits R.

Usage

```
mpi.exit()
mpi.quit(save = "no")
```

Arguments

`save`	the same argument as `quit` but default to "no".

Details

Normally, `mpi.finalize` is used to clean all MPI states. However, it will not detach the library Rmpi. To be more safe leaving MPI, `mpi.exit` not only calls `mpi.finalize` but also detaches the library Rmpi. This will make reload the library Rmpi impossible.

If leaving MPI and R altogether, one simply uses `mpi.quit`.

Value

`mpi.exit` always returns 1

Author(s)

Hao Yu

See Also

`mpi.finalize`

`mpi.finalize` *MPI_Finalize API*

Description

Terminates MPI execution environment.

Usage

```
mpi.finalize()
```

Arguments

None

Details

This routines must be called by each slave (master) before it exits. This call cleans all MPI state. Once `mpi.finalize` has been called, no MPI routine may be called. To be more safe leaving MPI, please use `mpi.exit` which not only calls `mpi.finalize` but also detaches the library Rmpi. This will make reload the library Rmpi impossible.

Value

Always return 1

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

mpi.exit

---

mpi.gather *MPI_Gather, MPI_Gatherv, MPI_Allgather, and MPI_Allgatherv APIs*

Description

mpi.gather and mpi.gatherv (vector variant) gather each member's message to the member specified by the argument root. The root member receives the messages and stores them in rank order. mpi.allgather and mpi.allgatherv are the same as mpi.gather and mpi.gatherv except that all members receive the result instead of just the root.

Usage

```
mpi.gather(x, type, rdata, root = 0, comm = 1)
mpi.gatherv(x, type, rdata, rcounts, root = 0, comm = 1)

mpi.allgather(x, type, rdata, comm = 1)
mpi.allgatherv(x, type, rdata, rcounts, comm = 1)
```

Arguments

x	data to be gathered. Must be the same type.
type	1 for integer, 2 for double, and 3 for character. Others are not supported.
rdata	the receive buffer. Must be the same type as the sender and big enough to include all message gathered.
rcounts	int vector specifying the length of each message.
root	rank of the receiver
comm	a communicator number

Details

For mpi.gather and mpi.allgather, the message to be gathered must be the same dim and the same type. The receive buffer can be prepared as either integer(size * dim) or double(size * dim), where size is the total number of members in a comm. For mpi.gatherv and mpi.allgatherv, the message to be gathered can have different dims but must be the same type. The argument rcounts records these different dims into an integer vector in rank order. Then the receive buffer can be prepared as either integer(sum(rcounts)) or double(sum(rcounts)).

Value

For `mpi.gather` or `mpi.gatherv`, it returns the gathered message for the root member. For other members, it returns what is in rdata, i.e., rdata (or rcounts) is ignored. For `mpi.allgather` or `mpi.allgatherv`, it returns the gathered message for all members.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.scatter`, `mpi.scatterv`.

Examples

```
#Need 3 slaves to run properly
#Or use mpi.spawn.Rslaves(nslaves=3)
#mpi.bcast.cmd(id <-mpi.comm.rank(.comm), comm=1)
#mpi.bcast.cmd(mpi.gather(letters[id],type=3,rdata=string(1)))

#mpi.gather(letters[10],type=3,rdata=string(4))

# mpi.bcast.cmd(x<-rnorm(id))
# mpi.bcast.cmd(mpi.gatherv(x,type=2,rdata=double(1),rcounts=1))
# mpi.gatherv(double(1),type=2,rdata=double(sum(1:3)+1),rcounts=c(1,1:3))

#mpi.bcast.cmd(out1<-mpi.allgatherv(x,type=2,rdata=double(sum(1:3)+1),
# rcounts=c(1,1:3)))
#mpi.allgatherv(double(1),type=2,rdata=double(sum(1:3)+1),rcounts=c(1,1:3))
```

`mpi.gather.Robj` *Extentions of MPI_Gather and MPI_Allgather APIs*

Description

`mpi.gather.Robj` gathers each member's object to the member specified by the argument `root`. The root member receives the objects as a list. `mpi.allgather.Robj` is the same as `mpi.gather.Robj` except that all members receive the result instead of just the root.

Usage

```
mpi.gather.Robj(obj=NULL, root = 0, comm = 1, ...)

mpi.allgather.Robj(obj=NULL, comm = 1)
```

Arguments

`obj`	data to be gathered. Could be different type.
`root`	rank of the gather
`comm`	a communicator number
`...`	optional arugments to `sapply`.

Details

Since sapply is used to gather all results, its default option "simplify=TRUE" is to simplify outputs. In some situations, this option is not desirable. Using "simplify=FALSE" as in the place of ... will tell sapply not to simplify and a list of outputs will be returned.

Value

For `mpi.gather.Robj`, it returns a list, the gathered message for the root member. For `mpi.allgatherv.Robj`, it returns a list, the gathered message for all members.

Author(s)

Hao Yu and Wei Xia

References

https://www.open-mpi.org/

See Also

`mpi.gather`, `mpi.allgatherv`.

Examples

```
#Assume that there are some slaves running
#mpi.bcast.cmd(id<-mpi.comm.rank())
#mpi.bcast.cmd(x<-rnorm(id))
#mpi.bcast.cmd(mpi.gather.Robj(x))
#x<-"test mpi.gather.Robj"
#mpi.gather.Robj(x)

#mpi.bcast.cmd(obj<-rnorm(id+10))
#mpi.bcast.cmd(nn<-mpi.allgather.Robj(obj))
#obj<-rnorm(5)
#mpi.allgather.Robj(obj)
#mpi.remote.exec(nn)
```

`mpi.get.count` *MPI_Get_count API*

Description

`mpi.get.count` finds the length of a received message.

Usage

```
mpi.get.count(type, status = 0)
```

Arguments

`type`	1 for integer, 2 for double, 3 for char.
`status`	a status number

Details

When `mpi.recv` is used to receive a message, the receiver buffer can be set to be bigger than the incoming message. To find the exact length of the received message, `mpi.get.count` is used to find its exact length. `mpi.get.count` must be called immediately after calling `mpi.recv` otherwise the status may be changed.

Value

the length of a received message.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.send`, `mpi.recv`, `mpi.get.sourcetag`, `mpi.probe`.

mpi.get.processor.name *MPI_Get_processor_name API*

Description

`mpi.get.processor.name` returns the host name (a string) where it is executed.

Usage

```
mpi.get.processor.name(short = TRUE)
```

Arguments

`short`	a logical.

Value

a base host name if `short` = `TRUE` and a full host name otherwise.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

mpi.get.sourcetag *Utility for finding the source and tag of a received message*

Description

`mpi.get.sourcetag` finds the source and tag of a received message.

Usage

```
mpi.get.sourcetag(status = 0)
```

Arguments

`status`	a status number

Details

When mpi.any.source and/or mpi.any.tag are used by mpi.recv or mpi.probe, one can use mpi.get.sourcetag to find who sends the message or with what a tag number. mpi.get.sourcetag must be called immediately after calling mpi.recv or mpi.probe otherwise the obtained information may not be right.

Value

2 dim int vector. The first integer is the source and the second is the tag.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

mpi.send, mpi.recv, mpi.probe, mpi.get.count

---

mpi.iapplyLB *(Load balancing) parallel apply with nonblocking features*

Description

(Load balancing) parallellapply and related functions.

Usage

```
mpi.iapplyLB(X, FUN, ..., apply.seq=NULL, comm=1, sleep=0.01)
mpi.iparApply(X, MARGIN, FUN, ..., job.num = mpi.comm.size(comm)-1,
                    apply.seq=NULL, comm=1, sleep=0.01)
mpi.iparLapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
    comm=1,sleep=0.01)
mpi.iparSapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
simplify=TRUE, USE.NAMES = TRUE, comm=1, sleep=0.01)
mpi.iparRapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
comm=1, sleep=0.01)
mpi.iparCapply(X, FUN, ..., job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
comm=1,sleep=0.01)
mpi.iparReplicate(n, expr, job.num=mpi.comm.size(comm)-1, apply.seq=NULL,
simplify = TRUE, comm=1,sleep=0.01)
mpi.iparMM(A, B, comm=1, sleep=0.01)
```

Arguments

`X`	an array or matrix.
`MARGIN`	vector specifying the dimensions to use.
`FUN`	a function.
`simplify`	logical; should the result be simplified to a vector or matrix if possible?
`USE.NAMES`	logical; if `TRUE` and if `X` is character, use `X` as `names` for the result unless it had names already.
`n`	number of replications.
`A`	a matrix
`B`	a matrix
`expr`	expression to evaluate repeatedly.
`job.num`	Total job numbers. If job numbers is bigger than total slave numbers (default value), a load balancing approach is used.
`apply.seq`	if reproducing the same computation (simulation) is desirable, set it to the integer vector .mpi.applyLB generated in previous computation (simulation).
`...`	optional arguments to `Fun`
`comm`	a communicator number
`sleep`	a sleep interval on master node (in sec)

Details

`mpi.iparApply`, `mpi.iparLapply`, `mpi.iparSapply`, `mpi.iparRapply`, `mpi.iparCapply`, `mpi.iparSapply`, `mi.iparReplicate`, and `mpi.iparMM` are nonblocking versions of `mpi.parApply`, `mpi.parLapply`, `mpi.parSapply`, `mpi.parRapply`, `mpi.parCapply`, `mpi.parSapply`, `mpi.parReplicate`, and `mpi.parMM` respectively. The main difference is that `mpi.iprobe` and `Sys.sleep` are used so that master node consumes almost no CPU cycles while waiting for slaves results. However, due to frequent wake/sleep cycles on master, those functions are not suitable for running small jobs on slave nodes. If anticipated computing time for each job is relatively long, e.g., minutes or hours, setting sleep to be 1 second or longer will further reduce load on master (only slightly).

See Also

`mpi.iapply`

`mpi.info.create` *MPI_Info_create, MPI_Info_free, MPI_Info_get, MPI_Info_set APIs*

Description

Many MPI APIs take an info argument for additional information passing. An info is an object which consists of many (key,value) pairs. Rmpi uses an internal memory to store an info object.

`mpi.info.create` creates a new info object.

`mpi.info.free` frees an info object and sets it to MPI_INFO_NULL.

`mpi.info.get` retrieves the value associated with key in an info.

`mpi.info.set` adds the key and value pair to info.

Usage

```
mpi.info.create(info = 0)
mpi.info.free(info = 0)
mpi.info.get(info = 0, key, valuelen)
mpi.info.set(info = 0, key, value)
```

Arguments

`info`	an info number.
`key`	a char (length 1).
`valuelen`	the length (nchar) of a key
`value`	a char (length 1).

Value

`mpi.info.create`, `mpi.info.free`, and `mpi.info.set` return 1 if success and 0 otherwise.

`mpi.info.get` returns the value (a char) for a given info and valuelen.

Author(s)

Hao Yu

See Also

`mpi.spawn.Rslaves`

`mpi.intercomm.merge` *MPI_Intercomm_merge API*

Description

Creates an intracommunicator from an intercommunicator

Usage

```
mpi.intercomm.merge(intercomm=2, high=0, comm=1)
```

Arguments

`intercomm`	an intercommunicator number
`high`	Used to order the groups of the two intracommunicators within comm when creating the new communicator
`comm`	a (intra)communicator number

Details

When master spawns slaves, an intercommunicator is created. To make communications (point-to-point or groupwise) among master and slaves, an intracommunicator must be created. `mpi.intercomm.merge` is used for that purpose. This is a collective call so all master and slaves call together. R slaves spawned by `mpi.spawn.Rslaves` should use `mpi.comm.get.parent` to get (set) an intercomm to a number followed by merging antercomm to an intracomm. One can use `mpi.comm.test.inter` to test if a communicator is an intercommunicator or not.

Value

1 if success. Otherwise 0.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.comm.test.inter`

`mpi.parSim` *Parallel Monte Carlo Simulation*

Description

Carry out parallel Monte Carlo simulation on R slaves spawned by using slavedaemon.R script and all executed results are returned back to master.

Usage

```
mpi.parSim(n=100, rand.gen=rnorm, rand.arg=NULL,statistic,
nsim=100, run=1, slaveinfo=FALSE, sim.seq=NULL, simplify=TRUE, comm=1, ...)
```

Arguments

`n`	sample size.
`rand.gen`	the random data generating function. See the details section
`rand.arg`	additional argument list to `rand.gen`.
`statistic`	the statistic function to be simulated. See the details section
`nsim`	the number of simulation carried on a slave which is counted as one slave job.
`run`	the number of looping. See the details section.
`slaveinfo`	if TRUE, the numbers of jobs finished by slaves will be displayed.

`sim.seq`	if reproducing the same simulation is desirable, set it to the integer vector .mpi.parSim generated in previous simulation.
`simplify`	logical; should the result be simplified to a vector or matrix if possible?
`comm`	a communicator number
`...`	optional arguments to `statistic`

Details

It is assumed that one simulation is carried out as `statistic(rand.gen(n))`, where `rand.gen(n)` can return any values as long as `statistic` can take them. Additional arguments can be passed to `rand.gen` by `rand.arg` as a list. Optional arguments can also be passed to `statistic` by the argument `...`.

Each slave job consists of `replicate(nsim,statistic(rand.gen(n)))`, i.e., each job runs `nsim` number of simulation. The returned values are transported from slaves to master.

The total number of simulation (TNS) is calculated as follows. Let slave.num be the total number of slaves in a `comm` and it is `mpi.comm.size(comm)-1`. Then TNS=slave.num*nsim*run and the total number of slave jobs is slave.num*run, where `run` is the number of looping from master perspective. If run=1, each slave will run one slave job. If run=2, each slave will run two slaves jobs on average, and so on.

The purpose of using `run` has two folds. It allows a tuneup of slave job size and total number of slave jobs to deal with two different cluster environments. On a cluster of slaves with equal CPU power, `run=1` is often enough. But if `nsim` is too big, one can set `run=2` and the slave jog size to be `nsim/2` so that TNS=slave.num*(nsim/2)*(2*run). This may improve R computation efficiency slightly. On a cluster of slaves with different CPU power, one can choose a big value of `run` and a small value of `nsim` so that master can dispatch more jobs to slaves who run faster than others. This will keep all slaves busy so that load balancing is achieved.

The sequence of slaves who deliver results to master are saved into `.mpi.parSim`. It keeps track which part of results done by which slaves. `.mpi.parSim` can be used to reproduce the same simulation result if the same seed is used and the argument `sim.seq` is equal to `.mpi.parSim`.

See the warning section before you use `mpi.parSim`.

Value

The returned values depend on values returned by `replicate` of `statistic(rand.gen(n))` and the total number of simulation (TNS). If `statistic` returns a single value, then the result is a vector of length TNS. If `statistic` returns a vector (list) of length `nrow`, then the result is a matrix of dimension `c(nrow, TNS)`.

Warning

It is assumed that a parallel RNG is used on all slaves. Run `mpi.setup.rngstream` on the master to set up a parallel RNG. Though `mpi.parSim` works without a parallel RNG, the quality of simulation is not guarantied.

`mpi.parSim` will automatically transfer `rand.gen` and `statistic` to slaves. However, any functions that `rand.gen` and `statistic` reply on but are not on slaves must be transfered to slaves before using `mpi.parSim`. You can use `mpi.bcast.Robj2slave` for that purpose. The same is applied to required packages or C/Fortran codes. You can use either `mpi.bcast.cmd` or put `required(package)` and/or `dyn.load(so.lib)` into `rand.gen` and `statistic`.

If `simplify` is TRUE, sapply style simplication is applied. Otherwise a list of length slave.num*run is returned.

Author(s)

Hao Yu

See Also

`mpi.setup.rngstream` `mpi.bcast.cmd` `mpi.bcast.Robj2slave`

`mpi.probe` *MPI_Probe and MPI_Iprobe APIs*

Description

`mpi.probe` uses the source and tag of incoming message to set a status. `mpi.iprobe` does the same except it is a nonblocking call, i.e., returns immediately.

Usage

```
mpi.probe(source, tag, comm = 1, status = 0)
mpi.iprobe(source, tag, comm = 1, status = 0)
```

Arguments

`source`	the source of incoming message or mpi.any.source() for any source.
`tag`	a tag number or mpi.any.tag() for any tag.
`comm`	a communicator number
`status`	a status number

Details

When `mpi.send` or other nonblocking sends are used to send a message, the receiver may not know the exact length before receiving it. `mpi.probe` is used to probe the incoming message and put some information into a status. Then the exact length can be found by using `mpi.get.count` to such a status. If the wild card `mpi.any.source` or `mpi.any.tag` are used, then one can use `mpi.get.sourcetag` to find the exact source or tag of a sender.

Value

`mpi.probe` returns 1 only after a matching message has been found.

`mpi.iproble` returns TRUE if there is a message that can be received; FALSE otherwise.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

mpi.send, mpi.recv, mpi.get.count

---

mpi.realloc *Find and increase the lengthes of MPI opaques comm, request, and status*

Description

`mpi.comm.maxsize`, `mpi.request.maxsize`, and `mpi.status.maxsize` find the lengthes of comm, request, and status arrayes respectively.

`mpi.realloc.comm`, `mpi.realloc.request` and `mpi.realloc.status` increase the lengthes of comm, request and status arrayes to `newmaxsize` respectively if `newmaxsize` is bigger than the original maximum size.

Usage

```
mpi.realloc.comm(newmaxsize)
mpi.realloc.request(newmaxsize)
mpi.realloc.status(newmaxsize)
mpi.comm.maxsize()
mpi.request.maxsize()
mpi.status.maxsize()
```

Arguments

`newmaxsize`	an integer.

Details

When **Rmpi** is loaded, Rmpi allocs comm array with size 10, request array with 10,000 and status array with 5,000. They should be enough in most cases. They use less than 150KB system memory. In rare case, one can use `mpi.realloc.comm`, `mpi.realloc.request` and `mpi.realloc.status` to increase them to bigger arrays.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

`mpi.reduce` *MPI_Reduce and MPI_Allreduce APIs*

Description

`mpi.reduce` and `mpi.allreduce` are global reduction operations. `mpi.reduce` combines each member's result, using the operation `op`, and returns the combined value(s) to the member specified by the argument `dest`. `mpi.allreduce` is the same as `mpi.reduce` except that all members receive the combined value(s).

Usage

```
mpi.reduce(x, type=2, op=c("sum","prod","max","min","maxloc","minloc"),
dest = 0, comm = 1)

mpi.allreduce(x, type=2, op=c("sum","prod","max","min","maxloc","minloc"),
comm = 1)
```

Arguments

`x`	data to be reduced. Must be the same dim and the same type for all members.
`type`	1 for integer and 2 for double. Others are not supported.
`op`	one of "sum", "prod", "max", "min", "maxloc", or "minloc".
`dest`	rank of destination
`comm`	a communicator number

Details

It is important that all members in a comm call either all `mpi.reduce` or all `mpi.allreduce` even though the master may not be in computation. They must provide exactly the same type and dim vectors to be reduced. If the operation "maxloc" or "minloc" is used, the combined vector is twice as long as the original one since the maximum or minimum ranks are included.

Value

`mpi.reduce` returns the combined value(s) to the member specified by `dest`. `mpi.allreduce` returns the combined values(s) to every member in a comm. The combined value(s) may be the summation, production, maximum, or minimum specified by the argument `op`. If the `op` is either "maxloc" or "minloc", then the maximum (minimum) value(s) along the maximum (minimum) rank(s) will be returned.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.gather`.

`mpi.remote.exec` *Remote Executions on R slaves*

Description

Remotely execute a command on R slaves spawned by using slavedaemon.R script and return all executed results back to master.

Usage

```
mpi.remote.exec(cmd, ..., simplify = TRUE, comm = 1, ret = TRUE)
```

Arguments

`cmd`	the command to be executed on R slaves
`...`	used as arguments to cmd (function command) for passing their (master) values to R slaves, i.e., if 'myfun(x)' will be executed on R slaves with 'x' as master variable, use mpi.remote.exec(cmd=myfun, x).
`simplify`	logical; should the result be simplified to a data.frame if possible?
`comm`	a communicator number.
`ret`	return executed results from R slaves if TRUE.

Details

Once R slaves are spawned by `mpi.spawn.Rslaves` with the slavedaemon.R script, they are waiting for instructions from master. One can use `mpi.bcast.cmd` to send a command to R slaves. However it will not return executed results. Hence `mpi.remote.exec` can be considered an extension to `mpi.bcast.cmd`.

Value

return executed results from R slaves if the argument `ret` is set to be TRUE. The value could be a data.frame if values (integer or double) from each slave have the same dimension. Otherwise a list is returned.

Warning

`mpi.remote.exec` may have difficult guessing invisible results on R slaves. Use `ret = FALSE` instead.

Author(s)

Hao Yu

See Also

`mpi.spawn.Rslaves`, `mpi.bcast.cmd`

Examples

```
#mpi.remote.exec(mpi.comm.rank())
# x=5
#mpi.remote.exec(rnorm,x)
```

`mpi.scatter` *MPI_Scatter and MPI_Scatterv APIs*

Description

`mpi.scatter` and `mpi.scatterv` are the inverse operations of `mpi.gather` and `mpi.gatherv` respectively.

Usage

```
mpi.scatter(x, type, rdata, root = 0,  comm = 1)
mpi.scatterv(x, scounts, type, rdata, root = 0, comm = 1)
```

Arguments

`x`	data to be scattered.
`type`	1 for integer, 2 for double, and 3 for character. Others are not supported.
`rdata`	the receive buffer. Must be the same type as the sender
`scounts`	int vector specifying the block length inside a message to be scattered to other members.
`root`	rank of the receiver
`comm`	a communicator number

Details

`mpi.scatter` scatters the message x to all members. Each member receives a portion of x with dim as length(x)/size in rank order, where size is the total number of members in a comm. So the receive buffer can be prepared as either integer(length(x)/size) or double(length(x)/size). For `mpi.scatterv`, scounts counts the portions (different dims) of x sent to each member. Each member needs to prepare the receive buffer as either integer(scounts[i]) or double(scounts[i]).

Value

For non-root members, `mpi.scatter` or `scatterv` returns the scattered message and ignores whatever is in x (or scounts). For the root member, it returns the portion belonging to itself.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.gather`, `mpi.gatherv`.

Examples

```
#Need 3 slaves to run properly
#Or run  mpi.spawn.Rslaves(nslaves=3)
#  num="123456789abcd"
#  scounts<-c(2,3,1,7)
#  mpi.bcast.cmd(strnum<-mpi.scatter(integer(1),type=1,rdata=integer(1),root=0))
#  strnum<-mpi.scatter(scounts,type=1,rdata=integer(1),root=0)
#  mpi.bcast.cmd(ans <- mpi.scatterv(string(1),scounts=0,type=3,rdata=string(strnum),
# root=0))
#  mpi.scatterv(as.character(num),scounts=scounts,type=3,rdata=string(strnum),root=0)
#  mpi.remote.exec(ans)
```

`mpi.scatter.Robj` *Extensions of MPI_ SCATTER and MPI_SCATTERV*

Description

`mpi.scatter.Robj` and `mpi.scatter.Robj2slave` are used to scatter a list to all members. They are more efficient than using any parallel apply functions.

Usage

```
mpi.scatter.Robj(obj = NULL, root = 0, comm = 1)
mpi.scatter.Robj2slave(obj, comm = 1)
```

Arguments

`obj`	a list object to be scattered from the root or master
`root`	rank of the scatter.
`comm`	a communicator number.

Details

mpi.scatter.Robj is an extension of mpi.scatter for scattering a list object from a sender (root) to everyone. mpi.scatter.Robj2slave scatters a list to all slaves.

Value

mpi.scatter.Robj for non-root members, returns the scattered R object. For the root member, it returns the portion belonging to itself. mpi.scatter.Robj2slave returns no value for the master and all slaves get their corresponding components in the list, i.e., the first slave gets the first component in the list.

Author(s)

Hao Yu and Wei Xia

See Also

mpi.scatter, mpi.gather.Robj,

Examples

```
#assume that there are three slaves running
#mpi.bcast.cmd(x<-mpi.scatter.Robj())

#xx <- list("master",rnorm(3),letters[2],1:10)
#mpi.scatter.Robj(obj=xx)

#mpi.remote.exec(x)

#scatter a matrix to slaves
#dat=matrix(1:24,ncol=3)
#splitmatrix = function(x, ncl) lapply(.splitIndices(nrow(x), ncl), function(i) x[i,])
#dat2=splitmatrix(dat,3)
#mpi.scatter.Robj2slave(dat2)
#mpi.remote.exec(dat2)
```

---

mpi.send *MPI_Send, MPI_Isend, MPI_Recv, and MPI_Irecv APIs*

Description

The pair mpi.send and mpi.recv are two most used blocking calls for point-to-point communications. An int, double or char vector can be transmitted from any source to any destination.

The pair mpi.isend and mpi.irecv are the same except that they are nonblocking calls.

Blocking and nonblocking calls are interchangeable, e.g., nonblocking sends can be matched with blocking receives, and vice-versa.

Usage

```
mpi.send(x, type, dest, tag,  comm = 1)
mpi.isend(x, type, dest, tag,  comm = 1, request=0)
mpi.recv(x, type, source, tag,  comm = 1, status = 0)
mpi.irecv(x, type, source, tag,  comm = 1, request = 0)
```

Arguments

`x`	data to be sent or received. Must be the same type for source and destination. The receive buffer must be as large as the send buffer.
`type`	1 for integer, 2 for double, and 3 for character. Others are not supported.
`dest`	the destination rank. Use `mpi.proc.null` for a fake destination.
`source`	the source rank. Use `mpi.any.source` for any source. Use `mpi.proc.null` for a fake source.
`tag`	non-negative integer. Use `mpi.any.tag` for any tag flag.
`comm`	a communicator number.
`request`	a request number.
`status`	a status number.

Details

The pair `mpi.send` (or `mpi.isend`) and `mpi.recv` (or `mpi.irecv`) must be used together, i.e., if there is a sender, then there must be a receiver. Any mismatch will result a deadlock situation, i.e., programs stop responding. The receive buffer must be large enough to contain an incoming message otherwise programs will be crashed. One can use `mpi.probe` (or mpi.iprobe) and `mpi.get.count` to find the length of an incoming message before calling `mpi.recv`. If `mpi.any.source` or `mpi.any.tag` is used in `mpi.recv`, one can use `mpi.get.sourcetag` to find out the source or tag of the received message. To send/receive an R object rather than an int, double or char vector, please use the pair `mpi.send.Robj` and `mpi.recv.Robj`.

Since `mpi.irecv` is a nonblocking call, `x` with enough buffer must be created before using it. Then use nonblocking completion calls such as `mpi.wait` or `mpi.test` to test if `x` contains data from sender.

If multiple nonblocking sends or receives are used, please use request number consecutively from 0. For example, to receive two messages from two slaves, try mpi.irecv(x,1,source=1,tag=0,comm=1,request=0) mpi.irecv(y,1,source=2,tag=0,comm=1,request=1) Then `mpi.waitany`, `mpi.waitsome` or `mpi.waitall` can be used to complete the operations.

Value

`mpi.send` and `mpi.isend` return no value. `mpi.recv` returns the int, double or char vector sent from `source`. However, `mpi.irecv` returns no value. See details for explanation.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

mpi.send.Robj, mpi.recv.Robj, mpi.probe, mpi.wait, mpi.get.count, mpi.get.sourcetag.

Examples

```
#on a slave
#mpi.send(1:10,1,0,0)

#on master
#x <- integer(10)
#mpi.irecv(x,1,1,0)
#x
#mpi.wait()
#x
```

---

mpi.send.Robj — *Extensions of MPI_Send and MPI_Recv APIs*

Description

`mpi.send.Robj` and `mpi.recv.Robj` are two extensions of `mpi.send` and `mpi.recv`. They are used to transmit a general R object from any source to any destination.

`mpi.isend.Robj` is a nonblocking version of `mpi.send.Robj`.

Usage

```
mpi.send.Robj(obj, dest, tag, comm = 1)
mpi.isend.Robj(obj, dest, tag, comm = 1, request=0)
mpi.recv.Robj(source, tag, comm = 1, status = 0)
```

Arguments

`obj`	an R object. Can be any R object.
`dest`	the destination rank.
`source`	the source rank or mpi.any.source() for any source.
`tag`	non-negative integer or mpi.any.tag() for any tag.
`comm`	a communicator number.
`request`	a request number.
`status`	a status number.

Details

`mpi.send.Robj` and `mpi.isend.Robj` use `serialize` to encode an R object into a binary char vector. It sends the message to the destination. The receiver decode the message back into an R object by using `unserialize`.

If `mpi.isend.Robj` is used, `mpi.wait` or `mpi.test` must be used to check the object has been sent.

Value

`mpi.send.Robj` or `mpi.isend.Robj` return no value. `mpi.recv.Robj` returns the the transmitted R object.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

See Also

`mpi.send`, `mpi.recv`, `mpi.wait`, `serialize`, `unserialize`,

`mpi.sendrecv` *MPI_Sendrecv and MPI_Sendrecv_replace APIs*

Description

`mpi.sendrecv` and `mpi.sendrecv.replace` execute blocking send and receive operations. Both of them combine the sending of one message to a destination and the receiving of another message from a source in one call. The source and destination are possibly the same. The send buffer and receive buffer are disjoint for `mpi.sendrecv`, while the buffers are not disjoint for `mpi.sendrecv.replace`.

Usage

```
mpi.sendrecv(senddata, sendtype, dest, sendtag, recvdata, recvtype,
source, recvtag, comm = 1, status = 0)

mpi.sendrecv.replace(x, type, dest, sendtag, source, recvtag,
comm = 1, status = 0)
```

Arguments

`x`	data to be sent or recieved. Must be the same type for source and destination.
`senddata`	data to be sent. May have different datatypes and lengths
`recvdata`	data to be recieved. May have different datatypes and lengths
`type`	type of the data to be sent or recieved. 1 for integer, 2 for double, and 3 for character. Others are not supported.
`sendtype`	type of the data to be sent. 1 for integer, 2 for double, and 3 for character. Others are not supported.
`recvtype`	type of the data to be recieved. 1 for integer, 2 for double, and 3 for character. Others are not supported.
`dest`	the destination rank. Use `mpi.proc.null` for a fake destination.
`source`	the source rank. Use `mpi.any.source` for any source. Use `mpi.proc.null` for a fake source.
`sendtag`	non-negative integer. Use `mpi.any.tag` for any tag flag.
`recvtag`	non-negative integer. Use `mpi.any.tag` for any tag flag.
`comm`	a communicator number.
`status`	a status number.

Details

The receive buffer must be large enough to contain an incoming message otherwise programs will be crashed. There is compatibility between send-receive and normal sends and receives. A message sent by a send-receive can be received by a regular receive and a send-receive can receive a message sent by a regular send.

Value

Returns the int, double or char vector sent from the send buffers.

Author(s)

Kris Chen

References

`https://www.open-mpi.org/`

See Also

`mpi.send.Robj`, `mpi.recv.Robj`, `mpi.probe`. `mpi.get.sourcetag`.

Examples

```
#mpi.sendrecv(as.integer(11:20),1,0,33,integer(10),1,0,33,comm=0)
#mpi.sendrecv.replace(seq(1,2,by=0.1),2,0,99,0,99,comm=0)
```

`mpi.setup.rngstream` *Setup parallel RNG on all slaves*

Description

`mpi.setup.rngstream` setups RNGstream on all slaves.

Usage

```
mpi.setup.rngstream(iseed=NULL, comm = 1)
```

Arguments

`iseed`	An integer to be supplied to `set.seed`, or NULL not to set reproducible seeds.
`comm`	A comm number.

Details

`mpi.setup.rngstream` can be run only on master node. It can be run later on with the same or different iseed.

Value

No value returned.

Author(s)

Hao Yu

`mpi.spawn.Rslaves` *Spawn and Close R Slaves*

Description

`mpi.spawn.Rslaves` spawns R slaves to those hosts automatically chosen by MPI or specific hosts assigned by the argument `hosts`. Those R slaves are running in R BATCH mode with a specific Rscript file. The default Rscript file "slavedaemon.R" provides interactive R slave environments.

`mpi.close.Rslaves` shuts down R slaves spawned by `mpi.spawn.Rslaves`.

`tailslave.log` view (from tail) R slave log files (assuming they are all in one working directory).

Usage

```
mpi.spawn.Rslaves(Rscript=system.file("slavedaemon.R", package="Rmpi"),
        nslaves=mpi.universe.size(), root = 0, intercomm = 2,
        comm = 1, hosts = NULL, needlog = TRUE, mapdrive=TRUE, quiet = FALSE,
nonblock=TRUE, sleep=0.1)

mpi.close.Rslaves(dellog = TRUE, comm = 1)
tailslave.log(nlines = 3, comm = 1)
```

Arguments

`Rscript`	an R script file used to run R in BATCH mode.
`nslaves`	number of slaves to be spawned.
`root`	the rank number of the member who spawns R slaves.
`intercomm`	an intercommunicator number
`comm`	a communicator number merged from an intercomm.
`hosts`	NULL or LAM node numbers to specify where R slaves to be spawned.
`needlog`	a logical. If TRUE, R BATCH outputs will be saved in log files. If FALSE, the outputs will send to /dev/null.
`mapdrive`	a logical. If TRUE and master's working dir is on a network, mapping network drive is attemped on remote nodes under windows platform.
`quiet`	a logical. If TRUE, do not print anything unless an error occurs.
`nonblock`	a logical. If TRUE, a nonblock procedure is used on all slaves so that they will consume none or little CPUs while waiting.
`sleep`	a sleep interval, used when nonblock=TRUE. Smaller sleep is, more response slaves are, more CPUs consume.
`dellog`	a logical specifying if R slave's log files are deleted or not.
`nlines`	number of lines to view from tail in R slave's log files.

Details

The R slaves that `mpi.spawn.Rslaves` spawns are really running a shell program which can be found in `system.file("Rslaves.sh",package="Rmpi")` which takes a Rscript file as one of its arguments. Other arguments are used to see if a log file (R output) is needed and how to name it. The master process id and the comm number, along with host names where R slaves are running are used to name these log files.

Once R slaves are successfully spawned, the mergers from an intercomm (default 'intercomm = 2') to a comm (default 'comm = 1') are automatically done on master and slaves (should be done if the default Rscript is replaced). If additional sets of R slaves are needed, please use 'comm = 3', 'comm = 4', etc to spawn them. At most a comm number up to 10 can be used. Notice that the default comm number for R slaves (using slavedaemon.R) is always 1 which is saved as .comm.

To spawn R slaves to specific hosts, please use the argument `hosts` with a list of those node numbers (an integer vector). Total node numbers along their host names can be found by using `lamhosts`. Notice that this is LAM-MPI specific.

Value

Unless `quiet` = `TRUE`, `mpi.spawn.Rslaves` prints to stdio how many slaves are successfully spawned and where they are running.

`mpi.close.Rslaves` return 1 if success and 0 otherwise.

`tailslave.log` returns last lines of R slave's log files.

Author(s)

Hao Yu

See Also

`mpi.comm.spawn`, `lamhosts`.

Examples

```
#mpi.spawn.Rslaves(nslaves=2)
#tailslave.log()
#mpi.remote.exec(rnorm(10))
#mpi.close.Rslaves()
```

`mpi.universe.size` *MPI_Universe_size API*

Description

`mpi.universe.size` returns the total number of CPUs available in a cluster. Some MPI implements may not have this MPI call available.

Usage

```
mpi.universe.size()
```

Arguments

None.

Author(s)

Hao Yu

References

https://www.open-mpi.org/

`mpi.wait` *Nonblocking completion operations*

Description

`mpi.cancel` cancels a nonblocking send or receive request.

`mpi.test.cancelled` tests if `mpi.cancel` cancels or not.

`wait`, `waitall`, `waitany`, and `waitsome` are used to complete nonblocking send or receive requests. They are not local.

`test`, `testall`, `testany`, and `testsome` are used to complete nonblocking send and receive requests. They are local.

Usage

```
mpi.cancel(request)
mpi.test.cancelled(status=0)
mpi.test(request, status=0)
mpi.testall(count)
mpi.testany(count, status=0)
mpi.testsome(count)
mpi.wait(request, status=0)
mpi.waitall(count)
mpi.waitany(count, status=0)
mpi.waitsome(count)
```

Arguments

`count`	total number of nonblocking operations.
`request`	a request number.
`status`	a status number.

Details

`mpi.wait` and `mpi.test` are used to complete a nonblocking send and receive request: use the same request number by `mpi.isend` or `mpi.irecv`. Once completed, the associated request is set to MPI_REQUEST_NULL and status contains information such as source, tag, and length of message.

If multiple nonblocking sends or receives are initiated, the following calls are more efficient. Make sure that request numbers are used consecutively as request=0, request=1, request=2, etc. In this way, the following calls can find request information in system memory.

`mpi.waitany` and `mpi.testany` are used to complete one out of several requests.

`mpi.waitall` and `mpi.testall` are used to complete all requests.

`mpi.waitsome` and `mpi.testsome` are used to complete all enabled requests.

Value

`mpi.cancel` returns no value.

`mpi.test.cancelled` returns TRUE if a nonblocking call is cancelled; FALSE otherwise.

`mpi.wait` returns no value. Instead status contains information that can be retrieved by `mpi.get.count` and `mpi.get.sourcetag`.

`mpi.test` returns TRUE if a request is complete; FALSE otherwise. If TRUE, it is the same as `mpi.wait`.

`mpi.waitany` returns which request (index) has been completed. In addition, status contains information that can be retrieved by `mpi.get.count` and `mpi.get.sourcetag`.

`mpi.testany` returns a list: index— request index; flag—TRUE if a request is complete; FALSE otherwise (index is no use in this case). If flag is TRUE, it is the same as `mpi.waitany`.

`mpi.waitall` returns no value. Instead statuses 0, 1, ..., count-1 contain corresponding information that can be retrieved by `mpi.get.count` and `mpi.get.sourcetag`.

`mpi.testall` returns TRUE if all requests are complete; FALSE otherwise. If TRUE, it is the same as `mpi.waitall`.

`mpi.waitsome` returns a list: count— number of requests that have been completed; indices—an integer vector of size count of those completed request numbers (in 0, 1 ,..., count-1). In addition, statuses 0, 1, ..., count-1 contain corresponding information that can be retrieved by `mpi.get.count` and `mpi.get.sourcetag`.

`mpi.testsome` is the same as `mpi.waitsome` except that count may be 0 and in this case indices is no use.

Author(s)

Hao Yu

References

`https://www.open-mpi.org/`

See Also

`mpi.isend`, `mpi.irecv`, `mpi.get.count`, `mpi.get.sourcetag`.

`string` *Internal functions*

Description

Internal and hidden functions used by other MPI functions.

`mpi.comm.is.null` is used to test if a comm is MPI_COMM_NULL (empty members).

`string` create a string (empty space character) buffer.

`.docall` a wrap to docall function.

`.mpi.worker.apply` apply like function used by workers.

`.mpi.worker.applyLB` apply like function used by workers (load balancing).

`.mpi.worker.exec` real execution by workers when using mpi.remote.exec.

`.mpi.worker.sim` real simulation by workers when using mpi.parSim.

`.type.index` identify input data type: integer, numeric, raw, or others.

`.simplify` simplify internal objects.

`.splitIndices` split parall apply jobs evenly.

`.onUnload` clean MPI when Rmpi is unloaded.

`.mpi.undefined` undefined mpi object.

`.force.type` force input data type object specified by type.

Usage

```
mpi.comm.is.null(comm)
string(length)
.docall(fun, args)
```

Arguments

`comm`	a communicator number.
`length`	length of a string.
`fun`	a function object.
`args`	arguments to function.

Value

`string` returns an empty character string.

Author(s)

Hao Yu

See Also

`mpi.spawn.Rslaves`

Index