

# Package ‘multicore’

February 14, 2012

**Version** 0.1-7

**Title** Parallel processing of R code on machines with multiple cores or CPUs

**Author** Simon Urbanek <Simon.Urbaneck@r-project.org>

**Maintainer** Simon Urbanek <Simon.Urbaneck@r-project.org>

**Depends** R (>= 2.0.0)

**Description** This package provides a way of running parallel computations in R on machines with multiple cores or CPUs. Jobs can share the entire initial workspace and it provides methods for results collection.

**License** GPL-2

**SystemRequirements** POSIX-compliant OS (essentially anything but Windows; some Windows variants are supported experimentally, your mileage may vary)

**OS\_type** unix

**URL** <http://www.rforge.net/multicore/>

**Repository** CRAN

**Date/Publication** 2011-09-08 04:11:11

## R topics documented:

children . . . . .	2
fork . . . . .	3
mclapply . . . . .	5
multicore . . . . .	6
parallel . . . . .	8
process . . . . .	10
pvec . . . . .	11
sendMaster . . . . .	12
signals . . . . .	13
<b>Index</b>	<b>15</b>

---

 children

*Functions for management of parallel children processes*


---

**Description**

children returns currently active children

readChild reads data from a given child process

selectChildren checks children for available data

readChildren checks children for available data and reads from the first child that has available data

sendChildStdin sends string (or data) to child's standard input

kill sends a signal to a child process

**Usage**

```
children(select)
readChild(child)
readChildren(timeout = 0)
selectChildren(children = NULL, timeout = 0)
sendChildStdin(child, what)
kill(process, signal = SIGINT)
```

**Arguments**

select	if omitted, all active children are returned, otherwise select should be a list of processes and only those from the list that are active will be returned.
child	child process (object of the class childProcess) or a process ID (pid)
timeout	timeout (in seconds, fractions supported) to wait before giving up. Negative numbers mean wait indefinitely (strongly discouraged as it blocks R and may be removed in the future).
children	list of child processes or a single child process object or a vector of process IDs or NULL. If NULL behaves as if all currently known children were supplied.
what	character or raw vector. In the former case elements are collapsed using the newline character. (But no trailing newline is added at the end!)
process	process (object of the class process) or a process ID (pid)
signal	signal to send (one of SIG. . . constants – see <a href="#">signals</a> – or a valid integer signal number)

**Value**

children returns a list of child processes (or an empty list)

readChild and readChildren return a raw vector with a "pid" attribute if data were available, integer vector of length one with the process ID if a child terminated or NULL if the child no longer exists (no children at all for readChildren).

`selectChildren` returns TRUE if the timeout was reached, FALSE if an error occurred (e.g. if the master process was interrupted) or an integer vector of process IDs with children that have data available.

`sendChildStdin` sends given content to the standard input (stdin) of the child process. Note that if the master session was interactive, it will also be echoed on the standard output of the master process (unless disabled). The function is vector-compatible, so you can specify more than one child as a list or a vector of process IDs.

`kill` returns TRUE.

### Warning

This is a very low-level API for expert use only. If you are interested in user-level parallel execution use [mclapply](#), [parallel](#) and friends instead.

### Author(s)

Simon Urbanek

### See Also

[fork](#), [sendMaster](#), [parallel](#), [mclapply](#)

---

fork

*Fork a copy of the current R process*

---

### Description

`fork` creates a new child process as a copy of the current R process

`exit` closes the current child process, informing the master process as necessary

### Usage

```
fork()
exit(exit.code = 0L, send = NULL)
```

### Arguments

`exit.code` process exit code. Currently it is not used by multicore, but other applications might. By convention 0 signifies clean exit, 1 an error.

`send` if not NULL send this data before exiting (equivalent to using [sendMaster](#))

## Details

The `fork` function provides an interface to the `fork` system call. In addition it sets up a pipe between the master and child process that can be used to send data from the child process to the master (see [sendMaster](#)) and child's `stdin` is re-mapped to another pipe held by the master process (see `link{sendChildStdin}`).

If you are not familiar with the `fork` system call, do not use this function since it leads to very complex inter-process interactions among the R processes involved.

In a nutshell `fork` spawns a copy (child) of the current process, that can work in parallel to the master (parent) process. At the point of forking both processes share exactly the same state including the workspace, global options, loaded packages etc. Forking is relatively cheap in modern operating systems and no real copy of the used memory is created, instead both processes share the same memory and only modified parts are copied. This makes `fork` an ideal tool for parallel processing since there is no need to setup the parallel working environment, data and code is shared automatically from the start.

It is *strongly discouraged* to use `fork` in GUI or embedded environments, because it leads to several processes sharing the same GUI which will likely cause chaos (and possibly crashes). Child processes should never use on-screen graphics devices.

## Value

`fork` returns an object of the class `childProcess` (to the master) and `masterProcess` (to the child). `exit` never returns

## Warning

This is a very low-level API for expert use only. If you are interested in user-level parallel execution use [mclapply](#), [parallel](#) and friends instead.

## Note

Windows operating system lacks the `fork` system call so it cannot be used with multicore.

## Author(s)

Simon Urbanek

## See Also

[parallel](#), [sendMaster](#)

## Examples

```
p <- fork()
if (inherits(p, "masterProcess")) {
  cat("I'm a child! ", Sys.getpid(), "\n")
  exit("I was a child")
}
cat("I'm the master\n")
unserialize(readChildren(1.5))
```

---

mclapply	<i>Parallel version of lapply</i>
----------	-----------------------------------

---

### Description

mclapply is a parallelized version of `lapply`, it returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

### Usage

```
mclapply(X, FUN, ..., mc.preschedule = TRUE, mc.set.seed = TRUE,  
         mc.silent = FALSE, mc.cores = getOption("cores"), mc.cleanup = TRUE)
```

### Arguments

<code>X</code>	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by <code>as.list</code> .
<code>FUN</code>	the function to be applied to each element of <code>X</code>
<code>...</code>	optional arguments to <code>FUN</code>
<code>mc.preschedule</code>	if set to <code>TRUE</code> then the computation is first divided to (at most) as many jobs as there are cores and then the jobs are started, each job possibly covering more than one value. If set to <code>FALSE</code> then one job is spawned for each value of <code>X</code> sequentially (if used with <code>mc.set.seed=FALSE</code> then random number sequences will be identical for all values). The former is better for short computations or large number of values in <code>X</code> , the latter is better for jobs that have high variance of completion time and not too many values of <code>X</code> .
<code>mc.set.seed</code>	if set to <code>TRUE</code> then each parallel process first sets its seed to something different from other processes. Otherwise all processes start with the same (namely current) seed.
<code>mc.silent</code>	if set to <code>TRUE</code> then all output on <code>stdout</code> will be suppressed for all parallel processes spawned ( <code>stderr</code> is not affected).
<code>mc.cores</code>	The number of cores to use, i.e. how many processes will be spawned (at most)
<code>mc.cleanup</code>	if set to <code>TRUE</code> then all children that have been spawned by this function will be killed (by sending <code>SIGTERM</code> ) before this function returns. Under normal circumstances <code>mclapply</code> waits for the children to deliver results, so this option usually has only effect when <code>mclapply</code> is interrupted. If set to <code>FALSE</code> then child processes are collected, but not forcefully terminated. As a special case this argument can be set to the signal value that should be used to kill the children instead of <code>SIGTERM</code> .

### Details

`mclapply` is a parallelized version of `lapply`, but there is an important difference: `mclapply` does not affect the calling environment in any way, the only side-effect is the delivery of the result (with the exception of a fall-back to `lapply` when there is only one core).

By default (`mc.preschedule=TRUE`) the input vector/list `X` is split into as many parts as there are cores (currently the values are spread across the cores sequentially, i.e. first value to core 1, second to core 2, ... (core + 1)-th value to core 1 etc.) and then one process is spawned to each core and the results are collected.

Due to the parallel nature of the execution random numbers are not sequential (in the random number sequence) as they would be in `lapply`. They are sequential for each spawned process, but not all jobs as a whole.

In addition, each process is running the job inside `try(..., silent=TRUE)` so if error occur they will be stored as `try-error` objects in the list.

Note: the number of file descriptors is usually limited by the operating system, so you may have trouble using more than 100 cores or so (see `ulimit -n` or similar in your OS documentation) unless you raise the limit of permissible open file descriptors (fork will fail with "unable to create a pipe").

### Value

A list.

### Author(s)

Simon Urbanek

### See Also

[parallel](#), [collect](#)

### Examples

```
mclapply(1:30, rnorm)
# use the same random numbers for all values
set.seed(1)
mclapply(1:30, rnorm, mc.preschedule=FALSE, mc.set.seed=FALSE)
# something a bit bigger - albeit still useless :P
unlist(mclapply(1:32, function(x) sum(rnorm(1e7))))
```

---

multicore

*multicore R package for parallel processing of R code*

---

### Description

*multicore* is an R package that provides functions for parallel execution of R code on machines with multiple cores or CPUs. Unlike other parallel processing methods all jobs share the full state of R when spawned, so no data or code needs to be initialized. The actual spawning is very fast as well since no new R instance needs to be started.

## Pivotal functions

[mclapply](#) - parallelized version of [lapply](#)

[pvec](#) - parallelization of vectorized functions

[parallel](#) and [collect](#) - functions to evaluate R expressions in parallel and collect the results.

## Low-level functions

Those function should be used only by experienced users understanding the interaction of the master (parent) process and the child processes (jobs) as well as the system-level mechanics involved.

See [fork](#) help page for the principles of forking parallel processes and system-level functions, [children](#) and [sendMaster](#) help pages for management and communication between the parent and child processes.

## Classes

*multicore* defines a few informal (S3) classes:

`process` is a list with a named entry `pid` containing the process ID.

`childProcess` is a subclass of `process` representing a child process of the current R process. A child process is a special process that can send messages to the parent process. The list may contain additional entries for IPC (more precisely file descriptors), however those are considered internal.

`masterProcess` is a subclass of `process` representing a handle that is passed to a child process by [fork](#).

`parallelJob` is a subclass of `childProcess` representing a child process created using the [parallel](#) function. It may (optionally) contain a `name` entry – a character vector of the length one as the name of the job.

## Options

By default functions that spawn jobs across cores use the "cores" option (see [options](#)) to determine how many cores (or CPUs) will be used (unless specified directly). If this option is not set, *multicore* uses by default as many cores as there are available. (Note: *cores* in this document refer to virtual cores. Modern CPUs can have more virtual cores than physical cores to accommodate simultaneous multithreading. For example, a machine with two quad-core Xeon W5590 processors has combined eight physical cores but 16 virtual cores. Also note that it is often beneficial to schedule more tasks than cores.)

The number of available cores is determined on startup using the (non-exported) `detectCores()` function. It should work on most commonly used unix systems (Mac OS X, Linux, Solaris and IRIX), but there is no standard way of determining the number of cores, so please contact me (with `sessionInfo()` output and the test) if you have tests for other platforms. If in doubt, use `multicore:::detectCores(all.tests=TRUE)` to see whether your platform is covered by one of the already existing tests. If *multicore* cannot determine the number of cores (the above returns NA), it will default to 8 (which should be fine for most modern desktop systems).

**Warning**

*multicore* uses the fork system call to spawn a copy of the current process which performs the computations in parallel. Modern operating systems use copy-on-write approach which makes this so appealing for parallel computation since only objects modified during the computation will be actually copied and all other memory is directly shared.

However, the copy shares everything including any user interface elements. This can cause havoc since let's say one window now suddenly belongs to two processes. Therefore *multicore* should be preferably used in console R and code executed in parallel may never use GUIs or on-screen devices.

An (experimental) way to avoid some such problems in some GUI environments (those using pipes or sockets) is to use `multicore:::closeAll()` in each child process immediately after it is spawned.

**Author(s)**

Simon Urbanek

**See Also**

[parallel](#), [mclapply](#), [fork](#), [sendMaster](#), [children](#) and [signals](#)

---

parallel

*Evaluate an expression asynchronously in a separate process*

---

**Description**

`parallel` starts a parallel process which evaluates the given expression.

`mcparrallel` is a synonym for `parallel` that can be used at top level if `parallel` is masked by other packages. It should not be used in other packages since it's just a shortcut for importing `multicore::parallel`.

`collect` collects results from parallel processes.

**Usage**

```
parallel(expr, name, mc.set.seed = FALSE, silent = FALSE)
mcparrallel(expr, name, mc.set.seed = FALSE, silent = FALSE)
collect(jobs, wait = TRUE, timeout = 0, intermediate = FALSE)
```

**Arguments**

<code>expr</code>	expression to evaluate (do <i>not</i> use any on-screen devices or GUI elements in this code)
<code>name</code>	an optional name (character vector of length one) that can be associated with the job.

<code>mc.set.seed</code>	if set to TRUE then the random number generator is seeded such that it is different from any other process. Otherwise it will be the same as in the current R session.
<code>silent</code>	if set to TRUE then all output on stdout will be suppressed (stderr is not affected).
<code>jobs</code>	list of jobs (or a single job) to collect results for. Alternatively jobs can also be an integer vector of process IDs. If omitted collect will wait for all currently existing children.
<code>wait</code>	if set to FALSE it checks for any results that are available within <code>timeout</code> seconds from now, otherwise it waits for all specified jobs to finish.
<code>timeout</code>	timeout (in seconds) to check for job results - applies only if <code>wait</code> is FALSE.
<code>intermediate</code>	FALSE or a function which will be called while collect waits for results. The function will be called with one parameter which is the list of results received so far.

### Details

`parallel` evaluates the `expr` expression in parallel to the current R process. Everything is shared read-only (or in fact copy-on-write) between the parallel process and the current process, i.e. no side-effects of the expression affect the main process. The result of the parallel execution can be collected using `collect` function.

`collect` function collects any available results from parallel jobs (or in fact any child process). If `wait` is TRUE then `collect` waits for all specified jobs to finish before returning a list containing the last reported result for each job. If `wait` is FALSE then `collect` merely checks for any results available at the moment and will not wait for jobs to finish. If `jobs` is specified, jobs not listed there will not be affected or acted upon.

Note: If `expr` uses low-level multicore functions such as [sendMaster](#) a single job can deliver results multiple times and it is the responsibility of the user to interpret them correctly. `collect` will return NULL for a terminating job that has sent its results already after which the job is no longer available.

### Value

`parallel` returns an object of the class `parallelJob` which is in turn a `childProcess`.

`collect` returns any results that are available in a list. The results will have the same order as the specified jobs. If there are multiple jobs and a job has a name it will be used to name the result, otherwise its process ID will be used.

### Author(s)

Simon Urbanek

### See Also

[mclapply](#), [sendMaster](#)

**Examples**

```

p <- parallel(1:10)
q <- parallel(1:20)
collect(list(p, q)) # wait for jobs to finish and collect all results

p <- parallel(1:10)
collect(p, wait=FALSE, 10) # will retrieve the result (since it's fast)
collect(p, wait=FALSE) # will signal the job as terminating
collect(p, wait=FALSE) # there is no such job

# a naive parallelized lapply can be created using parallel alone:
jobs <- lapply(1:10, function(x) parallel(rnorm(x), name=x))
collect(jobs)

```

---

process

*Function to query objects of the class process*


---

**Description**

processID returns the process IDs for the given processes. It raises an error if process is not an object of the class `process` or a list of such objects.

print methods shows the process ID and its class name.

**Usage**

```

processID(process)
## S3 method for class 'process'
print(x, ...)

```

**Arguments**

process	process (object of the class process) or a list of such objects.
x	process to print
...	ignored

**Value**

processID returns an integer vector containing the process IDs.

print returns NULL invisibly

**Author(s)**

Simon Urbanek

**See Also**

[fork](#)

pvec

*Parallelize a vector map function***Description**

pvec parallelizes the execution of a function on vector elements by splitting the vector and submitting each part to one core. The function must be a vectorized map, i.e. it takes a vector input and creates a vector output of exactly the same length as the input which doesn't depend on the partition of the vector.

**Usage**

```
pvec(v, FUN, ..., mc.set.seed = TRUE, mc.silent = FALSE,
      mc.cores = getOption("cores"), mc.cleanup = TRUE)
```

**Arguments**

v	vector to operate on
FUN	function to call on each part of the vector
...	any further arguments passed to FUN after the vector
mc.set.seed	if set to TRUE then each parallel process first sets its seed to something different from other processes. Otherwise all processes start with the same (namely current) seed.
mc.silent	if set to TRUE then all output on stdout will be suppressed for all parallel processes spawned (stderr is not affected).
mc.cores	The number of cores to use, i.e. how many processes will be spawned (at most)
mc.cleanup	flag specifying whether children should be terminated when the master is aborted (see description of this argument in <a href="#">mclapply</a> for details)

**Details**

pvec parallelizes  $FUN(x, \dots)$  where FUN is a function that returns a vector of the same length as x. FUN must also be pure (i.e., without side-effects) since side-effects are not collected from the parallel processes. The vector is split into nearly identically sized subvectors on which FUN is run. Although it is in principle possible to use functions that are not necessarily maps, the interpretation would be case-specific as the splitting is in theory arbitrary and a warning is given in such cases.

The major difference between pvec and [mclapply](#) is that mclapply will run FUN on each element separately whereas pvec assumes that  $c(FUN(x[1]), FUN(x[2]))$  is equivalent to  $FUN(x[1:2])$  and thus will split into as many calls to FUN as there are cores, each handling a subset vector. This makes it much more efficient than mclapply but requires the above assumption on FUN.

**Value**

The result of the computation - in a successful case it should be of the same length as v. If an error occurred or the function was not a map the result may be shorter and a warning is given.

**Note**

Due to the nature of the parallelization error handling does not follow the usual rules since errors will be returned as strings and killed child processes will show up simply as non-existent data. Therefore it is the responsibility of the user to check the length of the result to make sure it is of the correct size. pvec raises a warning if that is the case since it does not know whether such outcome is intentional or not.

**See Also**

[parallel](#), [mclapply](#)

**Examples**

```
x <- pvec(1:1000, sqrt)
stopifnot(all(x == sqrt(1:1000)))

# a common use is to convert dates to unix time in large datasets
# as that is an awfully slow operation
# so let's get some random dates first
dates <- sprintf('%04d-%02d-%02d', as.integer(2000+rnorm(1e5)),
                 as.integer(runif(1e5,1,12)), as.integer(runif(1e5,1,28)))

# this takes 4s on a 2.6GHz Mac Pro
system.time(a <- as.POSIXct(dates))

# this takes 0.5s on the same machine (8 cores, 16 HT)
system.time(b <- pvec(dates, as.POSIXct))

stopifnot(all(a == b))

# using mclapply for this is much slower because each value
# will require a separate call to as.POSIXct()
system.time(c <- unlist(mclapply(dates, as.POSIXct)))
```

---

sendMaster

*Sends data from the child to to the master process*

---

**Description**

sendMaster Sends data from the child to to the master process

**Usage**

sendMaster(what)

**Arguments**

what data to send to the master process. If what is not a raw vector, what will be serialized into a raw vector. Do NOT send an empty raw vector - it is reserved for internal use.

**Details**

Any child process (created by [fork](#) directly or by [parallel](#) indirectly) can send data to the parent (master) process. Usually this is used to deliver results from the parallel child processes to the master process.

**Value**

returns TRUE

**Author(s)**

Simon Urbanek

**See Also**

[parallel](#), [fork](#)

---

signals

*Signal constants (subset)*

---

**Description**

SIGALRM alarm clock  
SIGCHLD to parent on child stop or exit  
SIGHUP hangup  
SIGINFO information request  
SIGINT interrupt  
SIGKILL kill (cannot be caught or ignored)  
SIGQUIT quit  
SIGSTOP sendable stop signal not from tty  
SIGTERM software termination signal from kill  
SIGUSR1 user defined signal 1  
SIGUSR2 user defined signal 2

**Details**

See man `signal` in your OS for details. The above codes can be used in conjunction with the [kill](#) function to send signals to processes.

**Author(s)**

Simon Urbanek

**See Also**

[kill](#)

# Index

## \*Topic **interface**

- children, [2](#)
- fork, [3](#)
- mclapply, [5](#)
- multicore, [6](#)
- parallel, [8](#)
- process, [10](#)
- pvec, [11](#)
- sendMaster, [12](#)
- signals, [13](#)

as.list, [5](#)

childProcess (multicore), [6](#)

children, [2](#), [7](#), [8](#)

collect, [6](#), [7](#)

collect (parallel), [8](#)

exit (fork), [3](#)

fork, [3](#), [3](#), [7](#), [8](#), [10](#), [13](#)

kill, [13](#), [14](#)

kill (children), [2](#)

lapply, [5](#), [7](#)

masterProcess (multicore), [6](#)

mclapply, [3](#), [4](#), [5](#), [7–9](#), [11](#), [12](#)

mcpParallel (parallel), [8](#)

multicore, [6](#)

options, [7](#)

parallel, [3](#), [4](#), [6](#), [7](#), [8](#), [8](#), [12](#), [13](#)

parallelJob (multicore), [6](#)

print.process (process), [10](#)

process, [10](#), [10](#)

process (multicore), [6](#)

processID (process), [10](#)

pvec, [7](#), [11](#)

readChild (children), [2](#)

readChildren (children), [2](#)

selectChildren (children), [2](#)

sendChildStdin (children), [2](#)

sendMaster, [3](#), [4](#), [7–9](#), [12](#)

SIGALRM (signals), [13](#)

SIGCHLD (signals), [13](#)

SIGHUP (signals), [13](#)

SIGINFO (signals), [13](#)

SIGINT (signals), [13](#)

SIGKILL (signals), [13](#)

signals, [2](#), [8](#), [13](#)

SIGQUIT (signals), [13](#)

SIGSTOP (signals), [13](#)

SIGTERM (signals), [13](#)

SIGUSR1 (signals), [13](#)

SIGUSR2 (signals), [13](#)