

Package ‘multicore’

April 17, 2009

Version 0.1-3

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

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

Maintainer Simon Urbanek <Simon.Urbanek@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)

OS_type unix

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

Repository CRAN

Date/Publication 2009-02-03 20:03:43

R topics documented:

children	2
fork	3
mclapply	5
multicore	6
parallel	8
process	9
sendMaster	10
signals	11

Index	12
--------------	-----------

Description

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

Arguments

<code>child</code>	child process (object of the class <code>childProcess</code>) or a process ID (pid)
<code>timeout</code>	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).
<code>children</code>	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.
<code>what</code>	character or raw vector. In the former case elements are collapsed using the newline character. (But no trailing newline is added at the end!)
<code>process</code>	process (object of the class <code>process</code>) or a process ID (pid)
<code>signal</code>	signal to send (one of <code>SIG...</code> constants – see signals – 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

Parallel version of lapply

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,
```

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)

Details

mclapply is a parallelized version of `lapply`. 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)
```

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](#)
[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.

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 *multicomp* 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.

`mcpParallel` 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)
mcpParallel(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 <code>TRUE</code> 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 <code>TRUE</code> then all output on <code>stdout</code> will be suppressed (<code>stderr</code> is not affected).
<code>jobs</code>	list of jobs (or a single job) to collect results for. Alternatively <code>jobs</code> can also be an integer vector of process IDs. If omitted <code>collect</code> will wait for all currently existing children.
<code>wait</code>	if set to <code>FALSE</code> 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 <code>FALSE</code> .
<code>intermediate</code>	<code>FALSE</code> or a function which will be called while <code>collect</code> 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](#)

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, 9
- sendMaster, 10
- signals, 11

as.list, 5

childProcess (multicore), 6

children, 2, 6, 7

collect, 6

collect (parallel), 8

exit (fork), 3

fork, 3, 3, 6, 7, 10, 11

kill, 11

kill (children), 2

lapply, 5, 6

masterProcess (multicore), 6

mclapply, 3, 4, 5, 6, 7, 9

mcpipeline (parallel), 8

multicore, 6

options, 7

parallel, 3, 4, 6, 7, 8, 10, 11

parallelJob (multicore), 6

print.process (process), 9

process, 9, 9

process (multicore), 6

processID (process), 9

readChild (children), 2

readChildren (children), 2

selectChildren (children), 2

sendChildStdin (children), 2

sendMaster, 3, 4, 6, 7, 9, 10

SIGALRM (signals), 11

SIGCHLD (signals), 11

SIGHUP (signals), 11

SIGINFO (signals), 11

SIGINT (signals), 11

SIGKILL (signals), 11

signals, 2, 7, 11

SIGQUIT (signals), 11

SIGSTOP (signals), 11

SIGTERM (signals), 11

SIGUSR1 (signals), 11

SIGUSR2 (signals), 11