Package 'mcprogress'

July 22, 2025

Title Progress Bars and Messages for Parallel Processes

Version 0.1.1

Description Tools for monitoring progress during parallel processing. Lightweight package which acts as a wrapper around mclapply() and adds a progress bar to it in 'RStudio' or 'Linux' environments. Simply replace your original call to mclapply() with pmclapply(). A progress bar can also be displayed during parallelisation via the 'foreach' package. Also included are functions to safely print messages (including error messages) from within parallelised code, which can be useful for debugging parallelised R code.

BugReports https://github.com/myles-lewis/mcprogress/issues

URL https://github.com/myles-lewis/mcprogress

License GPL (>= 3)

Encoding UTF-8

Imports parallel

Suggests knitr, rmarkdown, rstudioapi

RoxygenNote 7.3.2

VignetteBuilder knitr

NeedsCompilation no

Author Myles Lewis [aut, cre] (ORCID: <https://orcid.org/0000-0001-9365-5345>)

Maintainer Myles Lewis <myles.lewis@qmul.ac.uk>

Repository CRAN

Date/Publication 2024-09-26 11:00:02 UTC

Contents

catchError *Catch error messages during parallel processing*

Description

Allows an expression to be wrapped in `try()` to catch error messages. Any error messages are printed to the console using `mcstop()`.

Usage

```
catchError(expr, ...)
```

Arguments

`expr`	An expression to be wrapped in `try()` to allow execution and catch error messages.
`...`	Optional objects to be tracked if you want to know state of objects at the point error messages are generated.

Value

Prints error messages during parallel processing. If there is no error, the result of the evaluated expression is returned.

See Also

`mcstop()`

cat_parallel *Versions of `cat()` and `message()` for parallel processing*

Description

Prints messages to the console using `echo` during to enable messages to be printed during parallel processing. Text is only printed if the Rstudio environment is detected.

Usage

```
cat_parallel(...)

message_parallel(...)
```

Arguments

`...`	zero or more objects which can be coerced to character and which are pasted together.

Value

Prints a message to the console. `cat_parallel()` uses no line feed, while `message_parallel()` always adds a newline.

`mcProgressBar` *Show progress bar during parallel processing*

Description

Uses `echo` to safely output a progress bar to Rstudio or Linux console during parallel processing.

Usage

```
mcProgressBar(
  val,
  len = 1L,
  cores = 1L,
  subval = NULL,
  title = "",
  spinner = FALSE,
  eta = TRUE,
  start = NULL,
  sensitivity = 0.01
)

closeProgress(start = NULL, title = "", eta = TRUE)
```

Arguments

`val`	Integer measuring progress
`len`	Total number of processes to be executed overall.
`cores`	Number of cores used for parallel processing.
`subval`	Optional subvalue ranging from 0 to 1 to enable granularity during long processes. Especially useful if `len` is small relative to `cores`.
`title`	Optional title for the progress bar.
`spinner`	Logical whether to show a spinner which moves when each core completes a process. More useful for relatively long processes where the length of time for each process to complete is variable. Not shown if `subval` is used. Can add significant overhead is `len` is large and each process is very fast.
`eta`	Logical whether to show estimated time to completion. `start` system time must be supplied with each call to `mcProgressbar()` in order to estimate the time to completion.
`start`	Used to pass the system time from the start of the call to show a total time elapsed. See the example below.
`sensitivity`	Determines maximum sensitivity with which to report progress for situations where `len` is large, to reduce overhead. Default 0.01 refers to 1%. Not used if `subval` is invoked.

Details

This package provides 2 main methods to show progress during parallelised code using `mclapply()`. If `X` (the list object looped over in a call to `mclapply()`) has many elements compared to the number of cores, then it is easiest to use `pmclapply()`. However, in some use cases the length of `X` is comparable to the number of cores and each process may take a long time. For example, machine learning applied to each of 8 folds on an 8-core machine will open 8 processes from the outset. Each process will often complete at roughly the same time. In this case `pmclapply()` is much less informative as it only shows completion at the end of 1 round of processes so it will go from 0% to 100%. In this example, if each process code is long and subprogress can be reported along the way, for example during nested loops, then `mcProgressBar()` provides a way to show the subprogress during the inner loop. The example below shows how to write code involving an outer call to `mclapply()` and an inner loop whose subprogress is tracked via calls to `mcProgressBar()`.

Technically only 1 process can be tracked. If `cores` is set to 4 and `subval` is invoked, then the 1st, 5th, 9th, 13th etc process is tracked. Subprogress of this process is computed as part of the number of blocks of processes required. ETA is approximate. As part of minimising overhead, it is only updated with each change in progress (i.e. each time a block of processes completes) or when subprogress changes. It is not updated by interrupt.

Value

No return value. Prints a progress bar to the console if called within an Rstudio or Linux environment.

Author(s)

Myles Lewis

See Also

`pmclapply()` `mclapply()`

Examples

```
if (Sys.info()["sysname"] != "Windows") {

## Example function with mclapply wrapped around another nested function
library(parallel)

my_fun <- function(x, cores) {
  start <- Sys.time()
  mcProgressBar(0, title = "my_fun")  # initialise progress bar
  res <- mclapply(seq_along(x), function(i) {
    # inner loop of calculation
    y <- 1:4
    inner <- lapply(seq_along(y), function(j) {
      Sys.sleep(0.2 + runif(1) * 0.1)
      mcProgressBar(val = i, len = length(x), cores, subval = j / length(y),
                    title = "my_fun")
      rnorm(4)
    })
```

```
    inner
  }, mc.cores = cores)
  closeProgress(start, title = "my_fun")  # finalise the progress bar
  res
}

res <- my_fun(letters[1:4], cores = 2)

## Example of long function
longfun <- function(x, cores) {
  start <- Sys.time()
  mcProgressBar(0, title = "longfun")  # initialise progress bar
  res <- mclapply(seq_along(x), function(i) {
    # long sequential calculation in parallel with 3 major steps
    Sys.sleep(0.2)
    mcProgressBar(val = i, len = length(x), cores, subval = 0.33,
                  title = "longfun")  # 33% complete
    Sys.sleep(0.2)
    mcProgressBar(val = i, len = length(x), cores, subval = 0.66,
                  title = "longfun")  # 66% complete
    Sys.sleep(0.2)
    mcProgressBar(val = i, len = length(x), cores, subval = 1,
                  title = "longfun")  # 100% complete
    return(rnorm(4))
  }, mc.cores = cores)
  closeProgress(start, title = "longfun")  # finalise the progress bar
  res
}

res <- longfun(letters[1:2], cores = 2)

}
```

`mcstop` *Stop and print error message during parallel processing*

Description

`mcstop()` is a multicore version of `stop()` which prints to the console using 'echo' during parallel commands such as `mclapply()`, to allow error messages to be more visible.

Usage

```
mcstop(...)
```

Arguments

`...`	Objects coerced to character and pasted together and printed to the console using echo.

Value

Prints an error message.

`pmclapply` *mclapply with progress bar*

Description

`pmclapply()` adds a progress bar to `mclapply()` in Rstudio or Linux environments using output to the console. It is designed to add very little overhead.

Usage

```
pmclapply(
  X,
  FUN,
  ...,
  progress = TRUE,
  spinner = FALSE,
  title = "",
  eta = TRUE,
  mc.preschedule = TRUE,
  mc.set.seed = TRUE,
  mc.silent = FALSE,
  mc.cores = getOption("mc.cores", 2L),
  mc.cleanup = TRUE,
  mc.allow.recursive = TRUE,
  affinity.list = NULL
)
```

Arguments

`X`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list()`.
`FUN`	the function to be applied via `mclapply()` to each element of `X` in parallel.
`...`	Optional arguments passed to `FUN`.
`progress`	Logical whether to show the progress bar.
`spinner`	Logical whether to show a spinner which moves each time a parallel process is completed. More useful if the length of time for each process to complete is variable.
`title`	Title for the progress bar.
`eta`	Logical whether to show estimated time to completion.
`mc.preschedule`, `mc.set.seed`, `mc.silent`, `mc.cleanup`, `mc.allow.recursive`, `affinity.list`	See `mclapply()`.

`mc.cores` The number of cores to use, i.e. at most how many child processes will be run simultaneously. The option is initialized from environment variable `MC_CORES` if set. Must be at least one, and parallelization requires at least two cores.

Details

This function can be used in an identical manner to `mclapply()`. It is ideal for use if the length of `X` is comparably > cores. As processes are spawned in a block and most code for each process completes at roughly the same time, processes move along in blocks as determined by `mc.cores`. To track progress, `pmclapply()` only tracks the nth process, where n=`mc.cores`. For example, with 4 cores, `pmclapply()` reports progress when the 4th, 8th, 12th, 16th etc process has completed. If the length of `X` is very large (e.g. in the 1000s), then the progress bar will only update for each 1% of progress in order to reduce overhead.

However, in some scenarios the length of `X` is comparable to the number of cores and each process may take a long time. For example, machine learning applied to each of 8 cross-validation folds on an 8-core machine will open 8 processes from the outset. Each process will often complete at roughly the same time. In this case `pmclapply()` is much less informative as it only shows completion at the end of 1 round of processes, so it will go from 0% straight to 100%. For this scenario, we recommend users use `mcProgressBar()` which allows more fine-grained reporting of subprogress from within a block of parallel processes.

ETA is approximate. As part of minimising overhead, it is only updated with each change in progress (i.e. each time a block of processes completes). It is not updated by interrupt.

Value

A list of the same length as `X` and named by `X`.

Author(s)

Myles Lewis

See Also

`mclapply()` `mcProgressBar()`

Examples

```
if (Sys.info()["sysname"] != "Windows") {

res <- pmclapply(letters[1:20], function(i) {
                 Sys.sleep(0.2 + runif(1) * 0.1)
                 setNames(rnorm(5), paste0(i, 1:5))
                 }, mc.cores = 2, title = "Working")

}
```

Index