Package 'bettermc'

October 12, 2022

Title Enhanced Fork-Based Parallelization

Version 1.1.2

Date 2021-08-02

Description Drop-in replacement for 'parallel::mclapply()' adding e.g. tracebacks, crash dumps, retries, condition handling, improved seeding, progress bars and faster inter process communication. Some of the internal functions are also exported for other use: 'etry()' (extended try), 'copy2shm()/allocate_from_shm()' (copy to and allocate from POSIX shared memory), 'char_map/map2char()' (split a character vector into its unique elements and a mapping on these) and various semaphore related functions.

Biarch true

License MIT + file LICENSE

URL https://github.com/gfkse/bettermc

BugReports https://github.com/gfkse/bettermc/issues

Encoding UTF-8

Depends R (>= 3.5.0)

Imports checkmate, parallel

RoxygenNote 7.2

Suggests progress, spelling, testthat (>= 2.1.0)

Language en-US

NeedsCompilation yes

Author Andreas Kersting [aut, cre, cph],
GfK SE [cph],
R Core team [ctb] ('etry()' and its print method borrow a lot from base R)

Maintainer Andreas Kersting <andreas.kersting@gfk.com>

Repository CRAN

Date/Publication 2021-08-02 08:50:02 UTC

R topics documented:

`char_map` *Split a Character Vector into its Unique Elements and a Mapping on These*

Description

This is implemented using a radix sort on the CHARSXPs directly, i.e. on the addresses of the strings in the global string cache. Hence, in contrast to `unique`, this function does not consider two strings equal which differ only in their encoding. Also, the order of the unique elements is undefined.

Usage

```
char_map(x)

map2char(map)
```

Arguments

`x`	a character vector. Long vectors are supported.
`map`	an object as returned by `char_map`.

Value

`char_map` returns an S3 object of class "char_map", which is a list with the following elements: (chars) the unique set of strings in `x` in undefined order, (idx) an integer (or - for long vectors - double) vector such that `map$chars[map$idx]` is identical to `x` (except maybe for attributes), (attributes) the attributes of x as a shallow copy of the corresponding pairlist.

`map2char` returns a character vector identical to `x`, including attributes.

Windows Support

Fully supported on Windows.

Lifecycle

[Stable]

Examples

```
x <- sample(letters, 100, replace = TRUE)
map <- char_map(x)
stopifnot(identical(x, map$chars[map$idx]))

names(x) <- 1:100
stopifnot(identical(x, map2char(char_map(x))))
```

---

compress_chars *Recursively Call* char_map/map2char *on a List*

Description

These originally internal functions are exported because they are also useful for reducing the size of e.g. a data frame before storing it to disk using saveRDS. This also improves the (de)serialization speed.

Usage

```
compress_chars(
  l,
  limit = 0L,
  compress_altreps = c("if_allocated", "yes", "no"),
  class = character()
)

uncompress_chars(l, class = character())
```

Arguments

l	an object, typically a list
limit	the minimum length of a character vector for char_map to be applied
compress_altreps	should a character vector be compressed if it is an ALTREP? The default "if_allocated" only does so if the regular representation was already created. This was chosen as the default because in this case is is the regular representation which would be serialized.
class	additional classes to set on the char_map-objects created by compress_chars. For uncompress_chars, only call map2char on those char_map-objects which additionally inherit from all these classes.

Value

For compress_chars, l, but with character vectors replaced by objects of class char_map. For uncompress_chars, l, but with all char_map-objects, which also inherit from all classes given in class, replaced by the original character vectors.

Windows Support

Fully supported on Windows.

Lifecycle

[Experimental]

Note

The object returned by `compress_chars` might be an invalid S3 object, e.g. if `l` is a data frame. These functions are intended to be called immediately before and after (de)serializing the object, i.e. compress -> serialize -> store/transfer -> de-serialize -> uncompress.

`copy2shm` *Copy to and Allocate from POSIX Shared Memory*

Description

Copy the data of a vector to a POSIX shared memory object and allocate from such.

Usage

```
copy2shm(x, name, overwrite = FALSE, copy = TRUE)

allocate_from_shm(obj, copy = obj$copy)
```

Arguments

`x`	a logical, integer, double, complex or raw vector, an S3 object based hereon or a factor. Long vectors are supported.
`name`	the name of the shared memory object to create. A portable name starts with a "/", followed by one or more (up to 253) characters, none of which are slashes. **Note:** on macOS the total length of the name must not exceed 31 characters.
`overwrite`	should an already existing shared memory object with the given name be overwritten? If `FALSE`, the copy fails if such an object already exists. **Note:** Due to bugs in the macOS implementation of POSIX shared memory, (as of now) only `FALSE` is supported.
`copy`	should the vector placed in shared memory be used directly (`FALSE`) by `allocate_from_shm` or rather a copy of it (`TRUE`)? `FALSE` is apparently faster (initially), but might require more memory in the long run (up to double what is normally required by such a vector): if we modify elements of such a vector, new memory (pages) will be allocated to hold these changed values. The original memory (pages) of the shared memory object will only be freed when the vector is garbage collected. If we initially copy the whole vector from shared memory to "regular" one, the former can be freed directly and the latter can be modified in place. **Note:** The value passed to `copy2shm` has no direct effect. It only sets the default value for `allocate_from_shm`, which can safely be changed. **Note 2:** `FALSE` is silently ignored on macOS.

obj — an object as returned by copy2shm, which was typically called in another process.

Value

copy2shm returns an S3 object of class "shm_obj", which is a list with the following elements: (name) the name of the shared memory object as given, (type) an integer specifying the type of x, (length) the number of elements in x as a double, (size) the size of the shared memory object in bytes as a double, (attributes) the attributes of x as a shallow copy of the corresponding pairlist, (copy) the default value for the copy argument passed to allocate_from_shm. **Note:** this function will not produce an error if an operation related directly to the copy to shared memory fails. In this case a character vector of length 1 containing the error message will be returned.

allocate_from_shm returns a vector. **Note:** this function cannot be called more than once on any obj, since it unlinks the shared memory object immediately after *trying* to open it. If copy = TRUE, the vector will be allocated using a custom allocator, but this is not guaranteed. As of now, vectors with less than two elements are allocated using R's default allocator. This implementational detail must not be relied on. If copy = FALSE, the custom allocator *privately* maps the shared memory object into the address space of the current process. In particular this means that changes made to this memory region by subsequently forked child processes are private to them: neither the parent nor a sibling process will see these changes. This is most probably what we want and expect.

Windows Support

Not supported on Windows.

Lifecycle

[Stable]

Note

See also the general notes on POSIX shared memory under mclapply.

Examples

```
if (tolower(Sys.info()[["sysname"]]) != "windows") {
  x <- runif(100)
  obj <- copy2shm(x, "/random")
  if (inherits(obj, "shm_obj")) {
    # copy2shm succeeded
    y <- allocate_from_shm(obj)
    stopifnot(identical(x, y))
  } else {
    # copy2shm failed -> print the error message
    print(obj)
  }
}
```

`etry` *Extended try*

Description

Extended version of `try` with support for tracebacks and crash dumps.

Usage

```
etry(
  expr,
  silent = FALSE,
  outFile = getOption("try.outFile", default = stderr()),
  max.lines = 100L,
  dump.frames = c("partial", "full", "full_global", "no")
)

## S3 method for class '`etry-error`'
print(
  x,
 max.lines = getOption("traceback.max.lines", getOption("deparse.max.lines", -1L)),
  ...
)
```

Arguments

`expr`	an R expression to try.
`silent`	logical: should the report of error messages be suppressed?
`outFile`	a connection, or a character string naming the file to print to (via `cat(*, file = outFile)`); used only if `silent` is false, as by default.
`max.lines`	for `etry`, the maximum number of lines to be *deparsed* per call. For `print`, the maximum number of lines to be *printed* per call. The default for the latter is unlimited.
`dump.frames`	should a crash dump (cf. `dump.frames`) be created in case of an error? The default "partial" omits the frames up to the call of `etry`. "full" and "no" do the obvious. "full_global" additionally also includes (a copy of) the global environment (cf. `include.GlobalEnv` argument of `dump.frames`).
`x`	an object of class "etry-error".
`...`	further arguments passed to or from other methods.

Value

For `etry`, the value of the expression if `expr` is evaluated without error, but an invisible object of class `c("etry-error", "try-error")` containing the error message if it fails. This object has three attributes: (condition) the error condition, (traceback) the traceback as returned by `.traceback`, (dump.frames) the crash dump which can be examined using `utils::debugger`.

Windows Support

Fully supported on Windows.

Lifecycle

[Stable]

`mclapply` *parallel::mclapply Wrapper for Better Performance, Error Handling, Seeding and UX*

Description

This wrapper for `parallel::mclapply` adds the following features:

- reliably detect if a child process failed with a fatal error or if it was killed.
- get tracebacks after non-fatal errors in child processes.
- retry on fatal and non-fatal errors.
- fail early after non-fatal errors in child processes.
- get crash dumps from failed child processes.
- capture output from child processes.
- track warnings, messages and other conditions signaled in the child processes.
- return results from child processes using POSIX shared memory to improve performance.
- compress character vectors in results to improve performance.
- reproducibly seed all function calls.
- display a progress bar.

Usage

```
mclapply(
  X,
  FUN,
  ...,
  mc.preschedule = TRUE,
  mc.set.seed = NA,
  mc.silent = FALSE,
  mc.cores = getOption("mc.cores", 2L),
  mc.cleanup = TRUE,
  mc.allow.recursive = TRUE,
  affinity.list = NULL,
  mc.allow.fatal = FALSE,
  mc.allow.error = FALSE,
  mc.retry = 0L,
  mc.retry.silent = FALSE,
```

```
  mc.retry.fixed.seed = FALSE,
  mc.fail.early = !(mc.allow.error || mc.retry != 0L),
  mc.dump.frames = c("partial", "full", "full_global", "no"),
  mc.dumpto = ifelse(interactive(), "last.dump", "file://last.dump.rds"),
  mc.stdout = c("capture", "output", "ignore"),
 mc.warnings = c("m_signal", "signal", "m_output", "output", "m_ignore", "ignore",
    "stop"),
 mc.messages = c("m_signal", "signal", "m_output", "output", "m_ignore", "ignore"),
  mc.conditions = c("signal", "ignore"),
  mc.compress.chars = TRUE,
  mc.compress.altreps = c("if_allocated", "yes", "no"),
  mc.share.vectors = getOption("bettermc.use_shm", TRUE),
  mc.share.altreps = c("no", "yes", "if_allocated"),
  mc.share.copy = TRUE,
  mc.shm.ipc = getOption("bettermc.use_shm", TRUE),
  mc.force.fork = FALSE,
  mc.progress = interactive()
)

crash_dumps  # environment with crash dumps created by mclapply (cf. mc.dumpto)
```

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 to (mclapply) each element of X or (mcmapply) in parallel to ....
...	For mclapply, optional arguments to FUN. For mcmapply and mcMap, vector or list inputs: see mapply.
mc.preschedule	if set to TRUE then the computation is first divided to (at most) as many jobs are there are cores and then the jobs are started, each job possibly covering more than one value. If set to FALSE then one job is forked for each value of X. The former is better for short computations or large number of values in X, the latter is better for jobs that have high variance of completion time and not too many values of X compared to mc.cores.
mc.set.seed	TRUE or FALSE are directly handled by parallel::mclapply. bettermc also supports two additional values: NA (the default) - seed every invocation of FUN differently but in a reproducible way based on the current state of the random number generator in the parent process. integerish value - call set.seed(mc.set.seed) in the parent and then continue as if mc.set.seed was NA. In both (NA- and integerish-) cases, the state of the random number generator, i.e. the object .Random.seed in the global environment, is restored at the end of the function to what it was when mclapply was called. If the random number generator is not yet initialized in the current session, it is initialized internally (by calling runif(1)) and the resulting state is what gets restored later. In particular, this means that the seed supplied as mc.set.seed does *not* seed the code following the call to mclapply. All this ensures that arguments like mc.cores,

mc.force.fork etc. can be adjusted without affecting the state of the RNG outside of mclapply.

mc.silent — if set to TRUE then all output on 'stdout' will be suppressed for all parallel processes forked ('stderr' is not affected).

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.

mc.cleanup — if set to TRUE then all children that have been forked by this function will be killed (by sending SIGTERM) before this function returns. Under normal circumstances mclapply waits for the children to deliver results, so this option usually has only effect when mclapply is interrupted. If set to FALSE then child processes are collected, but not forcefully terminated. As a special case this argument can be set to the number of the signal that should be used to kill the children instead of SIGTERM.

mc.allow.recursive — Unless true, calling mclapply in a child process will use the child and not fork again.

affinity.list — a vector (atomic or list) containing the CPU affinity mask for each element of X. The CPU affinity mask describes on which CPU (core or hyperthread unit) a given item is allowed to run, see mcaffinity. To use this parameter prescheduling has to be deactivated (mc.preschedule = FALSE).

mc.allow.fatal — should fatal errors in child processes make mclapply fail (FALSE, default) or merely trigger a warning (TRUE)?

TRUE returns objects of classes c("fatal-error", "try-error") for failed invocations. Hence, in contrast to parallel::mclapply, it is OK for FUN to return NULL.

mc.allow.fatal can also be NULL. In this case NULL is returned, which corresponds to the behavior of parallel::mclapply.

mc.allow.error — should non-fatal errors in FUN make mclapply fail (FALSE, default) or merely trigger a warning (TRUE)? In the latter case, errors are stored as class c("etry-error", "try-error") objects, which contain full tracebacks and potentially crash dumps (c.f. mc.dump.frames and etry).

mc.retry — abs(mc.retry) is the maximum number of retries of failed applications of FUN in case of both fatal and non-fatal errors. This is useful if we expect FUN to fail either randomly (e.g. non-convergence of a model) or temporarily (e.g. database connections). Additionally, if mc.retry <= -1, the value of mc.cores is gradually decreased with each retry to a minimum of 1 (2 if mc.force.fork = TRUE). This is useful if we expect failures due to too many parallel processes, e.g. the Linux Out Of Memory Killer sacrificing some of the child processes.

The environment variable "BMC_RETRY" indicates the current retry. A value of "0" means first try, a value of "1" first *re*try, etc.

mc.retry.silent — should the messages indicating both fatal and non-fatal failures during all but the last retry be suppressed (TRUE) or not (FALSE, default)?

`mc.retry.fixed.seed`
: should `FUN` for a particular element of `X` always be invoked with the same fixed seed (`TRUE`) or should a different seed be used on each try (`FALSE`, default)? Only effective if `mc.set.seed` is `NA` or a number.

`mc.fail.early`
: should we try to fail fast after encountering the first (non-fatal) error in `FUN`? Such errors will be recorded as objects of classes `c("fail-early-error", "try-error")`.

`mc.dump.frames`
: should we `dump.frames` on non-fatal errors in `FUN`? The default "partial" omits the frames (roughly) up to the call of `FUN`. See `etry` for the other options.

`mc.dumpto`
: where to save the result including the dumped frames if `mc.dump.frames != "no" & mc.allow.error == FALSE`? Either the name of the variable to create in the environment `bettermc::crash_dumps` or a path (prefixed with "file://") where to save the object.

`mc.stdout`
: how should standard output from `FUN` be handled? "capture" captures the output (in the child processes) and prints it in the parent process after *all* calls of `FUN` of the current try (cf. `mc.retry`), such that it can be captured, sinked etc. there. "output" *immediately* forwards the output to stdout of the parent; it cannot be captured, sinked etc. there. "ignore" means that the output is not forwarded in any way to the parent process. For consistency, all of this also applies if `FUN` is called directly from the main process, e.g. because `mc.cores = 1`.

`mc.warnings, mc.messages, mc.conditions`
: how should warnings, messages and other conditions signaled by `FUN` be handled? "signal" records all warnings/messages/conditions (in the child processes) and signals them in the master process after *all* calls of `FUN` of the current try (cf. `mc.retry`). "stop" converts warnings (only) into non-fatal errors in the child processes directly. "output" *immediately* forwards the messages to stderr of the parent; no condition is signaled in the parent process nor is the output capturable/sinkable. "ignore" means that the conditions are not forwarded in any way to the parent process. Options prefixed with "m" additionally try to invoke the "muffleWarning"/"muffleMessage" restart in the child process. Note that, if `FUN` is called directly from the main process, conditions might be signaled twice in the main process, depending on these arguments.

`mc.compress.chars`
: should character vectors be compressed using `char_map` before returning them from the child process? Can also be the minimum length of character vectors for which to enable compression. This generally increases performance because (de)serialization of character vectors is particularly expensive.

`mc.compress.altreps`
: should a character vector be compressed if it is an ALTREP? The default "if_allocated" only does so if the regular representation was already created. This was chosen as the default because in this case is is the regular representation which would be serialized.

`mc.share.vectors`
: should non-character `atomic` vectors, S3 objects based hereon and factors be returned from the child processes using POSIX shared memory (cf. `copy2shm`)? Can also be the minimum length of vectors for which to use shared memory. This generally increases performance because shared memory is a much faster

form of inter process communication compared to pipes and we do not need to serialize the vectors.

`mc.share.altreps`
: should a non-character vector be returned from the child process using POSIX shared memory if it is an ALTREP?

`mc.share.copy`
: should the parent process use a vector placed in shared memory due to `mc.share.vectors` directly (`FALSE`) or rather a copy of it (`TRUE`)? See `copy2shm` for the implications.

`mc.shm.ipc`
: should the results be returned from the child processes using POSIX shared memory (cf. `copy2shm`)?

`mc.force.fork`
: should it be ensured that `FUN` is always called in a forked child process, even if `length(X) == 1`? This is useful if we use forking to protect the main R process from fatal errors, memory corruption, memory leaks etc. occurring in `FUN`. This feature requires that `mc.cores >= 2` and also ensures that the effective value for `mc.cores` never drops to less than 2 as a result of `mc.retry` being negative.

`mc.progress`
: should a progress bar be printed to stderr of the parent process (package `progress` must be installed)?

Format

`crash_dumps` is an initially empty environment used to store the return values of `mclapply` (see below) including crash dumps in case of non-fatal errors and if `mc.dump.frames != "no" & mc.allow.error == FALSE`.

Value

`mclapply` returns a list of the same length as X and named by X. In case of fatal/non-fatal errors and depending on `mc.allow.fatal`/`mc.allow.error`/`mc.fail.early`, some of the elements might inherit from "fatal-error"/"etry-error"/"fail-early-error" and "try-error" or be `NULL`.

POSIX Shared Memory

The shared memory objects created by `mclapply` are named as follows (this may be subject to change): `/bmc_ppid_timestamp_idx_cntr` (e.g. `/bmc_21479_1601366973201_16_10`), with

ppid the process id of the parent process.

timestamp the time at which `mclapply` was invoked (in milliseconds since epoch; on macOS: seconds since epoch, due to its 31-character limit w.r.t. POSIX names).

idx the index of the current element of `X` (1-based).

cntr an internal counter (1-based) referring to all the objects created due to `mc.share.vectors` for the current value of `X`; a value of `0` is used for the object created due to `mc.shm.ipc`.

`bettermc::mclapply` does not err if copying data to shared memory fails. It will rather only print a message and return results the usual way.

POSIX shared memory has (at least) kernel persistence, i.e. it is not automatically freed due to process termination, except if the object is/was unlinked. `bettermc` tries hard to not leave any byte behind, but it could happen that unlinking is incomplete if the parent process is terminated while `bettermc::mclapply` is running.

On Linux you can generally inspect the (not-unlinked) objects currently stored in shared memory by listing the files under */dev/shm*.

(Linux) Size of POSIX Shared Memory

On Linux, POSIX shared memory is implemented using a *tmpfs* typically mounted under `/dev/shm`. If not changed by the distribution, the default size of it is 50% of physical RAM. It can be changed (temporarily) by remounting it with a different value for the *size* option, e.g. `mount -o "remount,size=90%" /dev/shm`.

(Linux) POSIX Shared Memory and Transparent Hugepage Support

When allocating a shared memory object of at least `getOption("bettermc.hugepage_limit", 104857600)` bytes of size (default is 100 MiB), we use `madvise(..., MADV_HUGEPAGE)` to request the allocation of (transparent) huge pages. For this to have any effect, the *tmpfs* used to implement POSIX shared memory on Linux (typically mounted under `/dev/shm`) must be (re)mounted with option *huge=advise*, i.e. `mount -o remount,huge=advise /dev/shm`. (The default is `huge=never`, but this might be distribution-specific.)

Windows Support

On Windows, otherwise valid values for various arguments are silently replaced as follows:

```
mc.cores <- 1L
mc.share.vectors <- Inf
mc.shm.ipc <- FALSE
mc.force.fork <- FALSE
mc.progress <- FALSE
if (mc.stdout == "output") mc.stdout <- "ignore"
if (mc.warnings == "output") mc.warnings <- "ignore"
if (mc.messages == "output") mc.messages <- "ignore"
```

Note: `parallel::mclapply` demands `mc.cores` to be exactly 1 on Windows; `bettermc::mclapply` sets it to 1 on Windows.

Furthermore, `parallel::mclapply` ignores the following arguments on Windows: `mc.preschedule`, `mc.silent`, `mc.cleanup`, `mc.allow.recursive`, `affinity.list`. For `mc.set.seed`, only the values `TRUE` and `FALSE` are ignored (by `parallel::mclapply`); the other values are handled by `bettermc::mclapply` as documented above.

Lifecycle

[Stable]

See Also

`copy2shm`, `char_map`, `parallel::mclapply`

sem *Named POSIX Semaphores*

Description

Named POSIX Semaphores

Usage

```
sem_open(name, create = FALSE, overwrite = FALSE, value = 0)

sem_post(sem)

sem_wait(sem)

sem_close(sem)

sem_unlink(name)
```

Arguments

name	the name of the semaphore. Consult man sem_overview for what makes a valid name.
create	should the semaphore be created if it currently does not exist?
overwrite	if create == TRUE, should we overwrite an already existing semaphore with the name (TRUE) or rather fail (FALSE).
value	the initial value of the semaphore (>= 0).
sem	an object as returned by sem_open.

Value

For sem_open, an object of class "sem", which is an external pointer to the POSIX semaphore. All other functions return NULL invisibly and are called for their side effects.

Windows Support

Not supported on Windows.

Lifecycle

[Experimental]

semv *POSIX-style System V Semaphores*

Description

Mimic the POSIX semaphore API with System V semaphores.

Usage

```
semv_open(value = 0)

semv_post(sid, undo = TRUE)

semv_wait(sid, undo = TRUE)

semv_unlink(sid)
```

Arguments

value	the initial value of the semaphore to create (>= 0).
sid	the semaphore id as returned by semv_open.
undo	should the operations (decrement/increment) on the semaphore be undone on process termination. This feature is probably the main reason to prefer System V semaphores to POSIX ones.

Value

For semv_open, an object of class "semv", which is an integer referring to the System V semaphore. All other functions return NULL invisibly and are called for their side effects.

Windows Support

Not supported on Windows.

Lifecycle

[Experimental]

Index