Package 'filearray'

January 28, 2022

Type Package

Title File-Backed Array for Out-of-Memory Computation

Version 0.1.3

Language en-US

Encoding UTF-8

License LGPL-3

URL http://dipterix.org/filearray/,
https://github.com/dipterix/filearray

BugReports https://github.com/dipterix/filearray/issues

Description Stores large arrays in files to avoid occupying large memories. Implemented with super fast gigabyte-level multi-threaded reading/writing via 'OpenMP'. Supports multiple non-character data types (double, float, complex, integer, logical, and raw).

Imports digest, methods, Rcpp

Suggests rmarkdown, knitr, testthat (>= 3.0.0)

RoxygenNote 7.1.2

LinkingTo BH, Rcpp

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation yes

Author Zhengjia Wang [aut, cre, cph]

Maintainer Zhengjia Wang <dipterix.wang@gmail.com>

Repository CRAN

Date/Publication 2022-01-28 03:20:02 UTC

R topics documented:

`apply` *Apply functions over file array margins (extended)*

Description

Apply functions over file array margins (extended)

Usage

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)

## S4 method for signature 'FileArray'
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

Arguments

`X`	a file array
`MARGIN`	scalar giving the subscripts which the function will be applied over. Current implementation only allows margin size to be one
`FUN`	the function to be applied
`...`	optional arguments to `FUN`
`simplify`	a logical indicating whether results should be simplified if possible

Value

See Section 'Value' in `apply`;

filearray *Create or load existing file arrays*

Description

Create or load existing file arrays

Usage

```
filearray_create(
  filebase,
  dimension,
  type = c("double", "float", "integer", "logical", "raw", "complex"),
  partition_size = NA,
  initialize = FALSE,
  ...
)

filearray_load(filebase, mode = c("readwrite", "readonly"))

filearray_checkload(
  filebase,
  mode = c("readonly", "readwrite"),
  ...,
  symlink_ok = TRUE
)
```

Arguments

filebase	a directory path to store arrays in the local file system. When creating an array, the path must not exist.
dimension	dimension of the array, at least length of 2
type	storage type of the array; default is 'double'. Other options include 'integer', 'logical', and 'raw'.
partition_size	positive partition size for the last margin, or NA to automatically guess; see 'Details'.
initialize	whether to initialize partition files; default is false for performance considerations. However, if the array is dense, it is recommended to set to true
...	additional headers to check used by filearray_checkload (see 'Details'). This argument is ignored by filearray_create, reserved for future compatibility.
mode	whether allows writing to the file; choices are 'readwrite' and 'readonly'.
symlink_ok	whether arrays with symbolic-link partitions can pass the test; this is usually used on bound arrays with symbolic-links; see filearray_bind;

Details

The file arrays partition out-of-memory array objects and store them separately in local file systems. Since R stores matrices/arrays in column-major style, file array uses the slowest margin (the last margin) to slice the partitions. This helps to align the elements within the files with the corresponding memory order. An array with dimension `100x200x300x400` has 4 margins. The length of the last margin is 400, which is also the maximum number of potential partitions. The number of partitions are determined by the last margin size divided by `partition_size`. For example, if the partition size is 1, then there will be 400 partitions. If the partition size if 3, there will be 134 partitions. The default partition sizes are determined internally following these priorities:

1. the file size of each partition does not exceed `1GB`
2. the number of partitions do not exceed 100

These two rules are not hard requirements. The goal is to reduce the numbers of partitions as much as possible.

The arguments `...` in `filearray_checkload` should be named arguments that provide additional checks for the header information. The check will fail if at least one header is not identical. For example, if an array contains header key-signature pair, one can use `filearray_checkload(...,key = signature)` to validate the signature. Note the comparison will be rigid, meaning the storage type of the headers will be considered as well. If the signature stored in the array is an integer while provided is a double, then the check will result in failure.

Value

A `FileArray-class` instance.

Author(s)

Zhengjia Wang

Examples

```
# Prepare
library(filearray)
filebase <- tempfile()
if(file.exists(filebase)){ unlink(filebase, TRUE) }

# create array
x <- filearray_create(filebase, dimension = c(200, 30, 8))
print(x)

# Assign values
x[] <- rnorm(48000)

# Subset
x[1,2,]

# load existing array
```

```
filearray_load(filebase)

x$set_header("signature", "tom")
filearray_checkload(filebase, signature = "tom")

## Not run:
# Trying to load with wrong signature
filearray_checkload(filebase, signature = "jerry")

## End(Not run)
```

`FileArray-class` *Definition of file array*

Description

S4 class definition of `FileArray`. Please use `filearray_create` and `filearray_load` to create instances.

Public Methods

`get_header(key, default = NULL)` Get header information; returns `default` if `key` is missing

`set_header(key, value)` Set header information; the extra headers will be stored in meta file. Please do not store large headers as they will be loaded into memory frequently.

`can_write()` Whether the array data can be altered

`create(filebase, dimension, type = "double", partition_size = 1)` Create a file array instance

`delete(force = FALSE)` Remove array from local file system and reset

`dimension()` Get dimension vector

`dimnames(v)` Set/get dimension names

`element_size()` Internal storage: bytes per element

`fill_partition(part, value)` Fill a partition with given scalar

`get_partition(part, reshape = NULL)` Get partition data, and reshape (if not null) to desired dimension

`expand(n)` Expand array along the last margin; returns true if expanded; if the `dimnames` have been assigned prior to expansion, the last dimension names will be filled with `NA`

`initialize_partition()` Make sure a partition file exists; if not, create one and fill with `NA`s or 0 (`type='raw'`)

`load(filebase, mode = c("readwrite", "readonly"))` Load file array from existing directory

`partition_path(part)` Get partition file path

`partition_size()` Get partition size; see `filearray`

`set_partition(part, value, ..., strict = TRUE)` Set partition value

`sexp_type()` Get data SEXP type; see R internal manuals

`show()` Print information

`type()` Get data type

`valid()` Check if the array is valid.

See Also

`filearray`

`filearray_bind` *Merge and bind homogeneous file arrays*

Description

The file arrays to be merged must be homogeneous: same data type, partition size, and partition length

Usage

```
filearray_bind(
  ...,
  .list = list(),
  filebase = tempfile(),
  symlink = FALSE,
  overwrite = FALSE,
  cache_ok = FALSE
)
```

Arguments

`..., .list`	file array instances
`filebase`	where to create merged array
`symlink`	whether to use `file.symlink`; if true, then partition files will be symbolic-linked to the original arrays, otherwise the partition files will be copied over. If you want your data to be portable, do not use symbolic-links. The default value is FALSE
`overwrite`	whether to overwrite when `filebase` already exists; default is false, which raises errors
`cache_ok`	see 'Details', only used if `overwrite` is true.

Details

The input arrays must share the same data type and partition size. The dimension for each partition should also be the same. For example an array `x1` has dimension $100x20x30$ with partition size `1`, then each partition dimension is $100x20x1$, and there are `30` partitions. `x1` can bind with another array of the same partition size. This means if `x2` has dimension $100x20x40$ and each partition size is `1`, then `x1` and `x2` can be merged.

If `filebase` exists and `overwrite` is `FALSE`, an error will always raise. If `overwrite=TRUE` and `cache_ok=FALSE`, then the existing `filebase` will be erased and any data stored within will be lost. If both `overwrite` and `cache_ok` are `TRUE`, then , before erasing `filebase`, the function validates the existing array header and compare the header signatures. If the existing header signature is the same as the array to be created, then the existing array will be returned. This `cache_ok` could be extremely useful when binding large arrays with `symlink=FALSE` as the cache might avoid moving files around. However, `cache_ok` should be enabled with caution. This is because only the header information will be compared, but the partition data will not be compared. If the existing array was generated from an old versions of the source arrays, but the data from the source arrays has been altered, then the `cache_ok=TRUE` is rarely proper as the cache is outdated.

The `symlink` option should be used with extra caution. Creating symbolic links is definitely faster than copying partition files. However, since the partition files are simply linked to the original partition files, changing to the input arrays will also affect the merged arrays, and vice versa; see 'Examples'. Also for arrays created from symbolic links, if the original arrays are deleted, while the merged arrays will not be invalidated, the corresponding partitions will no longer be accessible. Attempts to set deleted partitions will likely result in failure. Therefore `symlink` should be set to true when creating merged arrays are temporary for read-only purpose, and when speed and disk space is in consideration. For extended reading, please check `files` for details.

Value

A bound array in `'FileArray'` class.

Examples

```
partition_size <- 1
type <- "double"
x1 <- filearray_create(
    tempfile(), c(2,2), type = type,
    partition_size = partition_size)
x1[] <- 1:4
x2 <- filearray_create(
    tempfile(), c(2,1), type = type,
    partition_size = partition_size)
x2[] <- 5:6

y1 <- filearray_bind(x1, x2, symlink = FALSE)
y2 <- filearray_bind(x1, x2)

# y1 copies partition files, and y2 simply creates links
# if symlink is supported

y1[] - y2[]
```

```
# change x1
x1[1,1] <- NA

# y1 is not affected
y1[]

# y2 changes
y2[]
```

`filearray_threads` *Set or get file array threads*

Description

Will enable/disable multi-threaded reading or writing at `C++` level using 'OpenMP'.

Usage

```
filearray_threads(n, reset_after_fork = -1L)
```

Arguments

`n`	number of threads to set. If `n` is negative, then default to the number of cores that computer has.
`reset_after_fork`	when forking a cluster (in `'osx'` or `'linux'`), the number of threads will be set to one to avoid memory issues. Setting this to `1` will reset threads once forked clusters are shut down. Setting to `0` will disable reset, and `-1` to use last set values.

Value

An integer of current number of threads

`fmap` *Map multiple file arrays and save results*

Description

Advanced mapping function for multiple file arrays. `fmap` runs the mapping functions and stores the results in file arrays. `fmap2` stores results in memory. This feature is experimental. There are several constraints to the input. Failure to meet these constraints may result in undefined results, or even crashes. Please read Section 'Details' carefully before using this function.

Usage

```
fmap(x, fun, .y, .input_size = NA, .output_size = NA, ...)

fmap2(x, fun, .input_size = NA, .simplify = TRUE, ...)

fmap_element_wise(x, fun, .y, ..., .input_size = NA)
```

Arguments

x	a list of file arrays to map; each element of x must share the same dimensions.
fun	function that takes one list
.y	a file array object, used to save results
.input_size	number of elements to read from each array of x
.output_size	fun output vector length
...	other arguments passing to fun
.simplify	whether to apply simplify2array to the result

Details

Denote the first argument of fun as input, The length of input equals the length of x. The size of each element of input is defined by .input_size, except for the last loop. For example, given dimension of each input array as $10x10x10x10$, if .input_size=100, then length(input[[1]])=100. The total number of runs equals to length(x[[1]])/100. If .input_size=300, then length(input[[1]]) will be 300 except for the last run. This is because 10000 cannot be divided by 300. The element length of the last run will be 100.

The returned variable length of fun will be checked by .output_size. If the output length exceed .output_size, an error will be raised.

Please make sure that length(.y)/length(x[[1]]) equals to .output_size/.input_size.

For fmap_element_wise, the input[[1]] and output length must be the consistent.

Value

File array instance .y

Examples

```
set.seed(1)
x1 <- filearray_create(tempfile(), dimension = c(100,20,3))
x1[] <- rnorm(6000)
x2 <- filearray_create(tempfile(), dimension = c(100,20,3))
x2[] <- rnorm(6000)

# Add two arrays
output <- filearray_create(tempfile(), dimension = c(100,20,3))
fmap(list(x1, x2), function(input){
```

```
    input[[1]] + input[[2]]
}, output)

# check
range(output[] - (x1[] + x2[]))

output$delete()

# Calculate the maximum of x1/x2 for every 100 elements
output <- filearray_create(tempfile(), dimension = c(20,3))
fmap(list(x1, x2), function(input){
    max(input[[1]] / input[[2]])
}, output, .input_size = 100, .output_size = 1)

# check
range(output[] - apply(x1[] / x2[], c(2,3), max))

output$delete()

# A large array example
if(interactive()){
    x <- filearray_create(tempfile(), dimension = c(287, 100, 301, 4))
    dimnames(x) <- list(
        Trial = 1:287,
        Marker = 1:100,
        Time = 1:301,
        Location = 1:4
    )

    for(i in 1:4){
        x[,,,i] <- runif(8638700)
    }
    # Step 1:
    # for each location, trial, and marker, calibrate (baseline)
    # according to first 50 time-points

    output <- filearray_create(tempfile(), dimension = dim(x))

    # baseline-percentage change
    fmap(
        list(x),
        function(input){
            # get locational data
            location_data <- input[[1]]
            dim(location_data) <- c(287, 100, 301)

            # collapse over first 50 time points for
            # each trial, and marker
            baseline <- apply(location_data[,,1:50], c(1,2), mean)

            # calibrate
            calibrated <- sweep(location_data, c(1,2), baseline,
                                FUN = function(data, bl){
```

```
                                    (data / bl - 1) * 100
                                })
            return(calibrated)
        },

        .y = output,

        # input dimension is 287 x 100 x 301 for each location
        .input_size = 8638700,

        # output dimension is 287 x 100 x 301
        .output_size = 8638700
    )

    # cleanup
    x$delete()

}

# cleanup
x1$delete()
x2$delete()
output$delete()
```

`fwhich` *A generic function of* `which` *that is* `'FileArray'` *compatible*

Description

A generic function of `which` that is `'FileArray'` compatible

Usage

```
fwhich(x, val, arr.ind = FALSE, ...)

## Default S3 method:
fwhich(x, val, arr.ind = FALSE, ...)

## S3 method for class 'FileArray'
fwhich(x, val, arr.ind = FALSE, ...)
```

Arguments

`x`	any R vector, matrix, array or file-array
`val`	values to find
`arr.ind`	logical; should array indices be returned when `x` is an array?
`...`	further passed to `which` or `arrayInd`

Value

The indices of x elements that are listed in val.

Examples

```
# Default case
x <- array(1:27, rep(3,3))
fwhich(x, c(4,5))

# file-array case
arr <- filearray_create(tempfile(), dim(x))
arr[] <- x
fwhich(arr, c(4,5))
```

---

mapreduce *A map-reduce method to iterate blocks of file-array data with little memory usage*

Description

A map-reduce method to iterate blocks of file-array data with little memory usage

Usage

```
mapreduce(x, map, reduce, ...)

## S4 method for signature 'FileArray,ANY,`function`'
mapreduce(x, map, reduce, buffer_size = NA, ...)

## S4 method for signature 'FileArray,ANY,`NULL`'
mapreduce(x, map, reduce, buffer_size = NA, ...)

## S4 method for signature 'FileArray,ANY,missing'
mapreduce(x, map, reduce, buffer_size = NA, ...)
```

Arguments

x	a file array object
map	mapping function that receives 3 arguments; see 'Details'
reduce	NULL, or a function that takes a list as input
...	passed to other methods
buffer_size	control how we split the array; see 'Details'

Details

When handling out-of-memory arrays, it is recommended to load a block of array at a time and execute on block level. See `apply` for a implementation. When an array is too large, and when there are too many blocks, this operation will become very slow if computer memory is low. This is because the R will perform garbage collection frequently. Implemented in `C++`, `mapreduce` creates a buffer to store the block data. By reusing the memory over and over again, it is possible to iterate through the array with minimal garbage collections. Many statistics, including `min`, `max`, `sum`, `mean`, ... These statistics can be calculated in this way efficiently.

The function `map` contains three arguments: `data` (mandate), `size` (optional), and `first_index` (optional). The `data` is the buffer, whose length is consistent across iterations. `size` indicates the effective size of the buffer. If the partition size is not divisible by the buffer size, only first `size` elements of the data are from array, and the rest elements will be `NA`. This situation could only occurs when `buffer_size` is manually specified. By default, all of `data` should belong to arrays. The last argument `first_index` is the index of the first element `data[1]` in the whole array. It is useful when positional data is needed.

The buffer size, specified by `buffer_size` is an additional optional argument in `...`. Its default is `NA`, and will be calculated automatically. If manually specified, a large buffer size would be desired to speed up the calculation. The default buffer size will not exceed $nThreads x 2MB$, where `nThreads` is the number of threads set by `filearray_threads`. When partition length cannot be divided by the buffer size, instead of trimming the buffer, `NA`s will be filled to the buffer, passed to `map` function; see previous paragraph for treatments.

The function `mapreduce` ignores the missing partitions. That means if a partition is missing, its data will not be read nor passed to `map` function. Please run `x$initialize_partition()` to make sure partition files exist.

Value

If `reduce` is `NULL`, return mapped results, otherwise return reduced results from `reduce` function

Examples

```
x <- filearray_create(tempfile(), c(100, 100, 10))
x[] <- rnorm(1e5)

## calculate summation
# identical to sum(x[]), but is more feasible in large cases

mapreduce(x, map = function(data, size){
    # make sure `data` is all from array
    if(length(data) != size){
        data <- data[1:size]
    }
    sum(data)
}, reduce = function(mapped_list){
    do.call(sum, mapped_list)
})
```

```
## Find elements are less than -3
positions <- mapreduce(
    x,
    map = function(data, size, first_index) {
        if (length(data) != size) {
            data <- data[1:size]
        }
        which(data < -3) + (first_index - 1)
    },
    reduce = function(mapped_list) {
        do.call(c, mapped_list)
    }
)

if(length(positions)){
    x[[positions[1]]]
}
```

S3-filearray *'S3' methods for 'FileArray'*

Description

These are 'S3' methods for 'FileArray'

Usage

```
## S3 method for class 'FileArray'

  x[
  ...,
  drop = TRUE,
  reshape = NULL,
  strict = TRUE,
  dimnames = TRUE,
  split_dim = 0
]

## S3 replacement method for class 'FileArray'
x[...] <- value

## S3 method for class 'FileArray'
x[[i]]

## S3 method for class 'FileArray'
as.array(x, reshape = NULL, drop = FALSE, ...)
```

```
## S3 method for class 'FileArray'
dim(x)

## S3 method for class 'FileArray'
dimnames(x)

## S3 replacement method for class 'FileArray'
dimnames(x) <- value

## S3 method for class 'FileArray'
length(x)

## S3 method for class 'FileArray'
max(x, na.rm = FALSE, ...)

## S3 method for class 'FileArray'
min(x, na.rm = FALSE, ...)

## S3 method for class 'FileArray'
range(x, na.rm = FALSE, ...)

## S3 method for class 'FileArray'
sum(x, na.rm = FALSE, ...)

## S3 method for class 'FileArray'
subset(x, ..., drop = FALSE)
```

Arguments

x	a file array
drop	whether to drop dimensions; see topic Extract
reshape	a new dimension to set before returning subset results; default is NULL (use default dimensions)
strict	whether to allow indices to exceed bound; currently only accept TRUE
dimnames	whether to preserve dimnames
split_dim	internally used; split dimension and calculate indices to manually speed up the subset; value ranged from 0 to size of dimension minus one.
value	value to substitute or set
i, ...	index set, or passed to other methods
na.rm	whether to remove NA values during the calculation

Functions

- [.FileArray: get element by position
- [<-.FileArray: assign array values

- `[[.FileArray`: get element by index
- `as.array.FileArray`: converts file array to native array in R
- `dim.FileArray`: get dimensions
- `dimnames.FileArray`: get dimension names
- `dimnames<-.FileArray`: set dimension names
- `length.FileArray`: get array length
- `max.FileArray`: get max value
- `min.FileArray`: get min value
- `range.FileArray`: get value range
- `sum.FileArray`: get summation
- `subset.FileArray`: get subset file array with formulae

`typeof` *The type of a file array (extended)*

Description

The type of a file array (extended)

Usage

```
typeof(x)

## S4 method for signature 'FileArray'
typeof(x)
```

Arguments

`x`	any file array

Value

A character string. The possible values are `"double"`, `"integer"`, `"logical"`, and `"raw"`

Index