Working with large arrays in R
A look at HDF5Array/RleArray/DelayedArray objects

Hervé Pagès
hpages.on.github@gmail.com

Bioconductor conference
Boston

July 2017
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
Motivation and challenges

R ordinary **matrix** or **array** is not suitable for big datasets:

- 10x Genomics dataset (single cell experiment): 30,000 genes x 1.3 million cells = 36.5 billion values
- in an ordinary integer matrix ==> 136G in memory!

Need for alternative containers:

- but at the same time, the object should be (almost) as easy to manipulate as an ordinary matrix or array
- **standard R matrix/array API**: `dim`, `dimnames`, `t`, `is.na`, `==`, `+`, `log`, `cbind`, `max`, `sum`, `colSums`, etc...
- not limited to 2 dimensions ==> also support arrays of arbitrary number of dimensions

2 approaches: **in-memory data** vs **on-disk data**
Motivation and challenges

In-memory data

- A 30k x 1.3M matrix might still fit in memory if the data can be efficiently compressed.
- Example: sparse data (small percentage of nonzero values) \(\Rightarrow\) sparse representation (storage of nonzero values only).
- Example: data with long runs of identical values \(\Rightarrow\) RLE compression (Run Length Encoding).
- Choose the smallest type to store the values: raw (1 byte) < integer (4 bytes) < double (8 bytes).
- If using RLE compression:
  - Choose the best orientation to store the values: by row or by column (one might give better compression than the other).
  - Store the data by chunk \(\Rightarrow\) opportunity to pick up best type and best orientation on a chunk basis (instead of for the whole data).
- Size of 30k x 1.3M matrix in memory can be reduced from 136G to 16G!
Motivation and challenges

Examples of in-memory containers

**dgCMatrix** container from the *Matrix* package:

- sparse matrix representation
- nonzero values stored as `double`

**RleArray** and **RleMatrix** containers from the *DelayedArray* package:

- use RLE compression
- arbitrary number of dimensions
- type of values: any R atomic type (integer, double, logical, complex, character, and raw)
Motivation and challenges

On-disk data

However...

- if data is too big to fit in memory (even after compression) \(\implies\) must use on-disk representation

- challenge: should still be (almost) as easy to manipulate as an ordinary matrix! (standard R matrix/array API)
Motivation and challenges

Examples of on-disk containers

Direct manipulation of an **HDF5 dataset** via the *rhdf5* API. Low level API!

**HDF5Array** and **HDF5Matrix** containers from the *HDF5Array* package:

Provide access to the HDF5 dataset via an API that mimics the standard R matrix/array API
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
The "airway" dataset

```r
library(airway)
data(airway)
m <- unname(assay(airway))
dim(m)
## [1] 63677  8
typeof(m)
## [1] "integer"
```

```r
head(m, n=4)
## [1,]  679  448  873  408 1138 1047  770  572
## [2,]   0   0   0   0   0   0   0   0
## [3,]  467  515  621  365  587  799  417  508
## [4,]  260  211  263  164  245  331  233  229
```

```r
tail(m, n=4)
## [63674,]  0   0   0   0   0   0   0   0
## [63675,]  0   0   0   0   0   0   0   0
## [63676,]  0   1   0   0   0   0   0   0
## [63677,]  0   0   0   0   1   0   0   0

sum(m != 0) / length(m)
## [1] 0.3889591
```
Memory footprint

dgCMatrix vs RleMatrix vs HDF5Matrix

```r
library(lobstr)  # for obj.size()
obj_size(m)
## 2.04 MB

library(Matrix)
obj_size(as(m, "dgCMatrix"))
## 2.38 MB

library(DelayedArray)
obj_size(as(m, "RleMatrix"))
## 2.22 MB

obj_size(as(t(m), "RleMatrix"))
## 1.74 MB

library(HDF5Array)
obj_size(as(m, "HDF5Matrix"))
## 2.38 kB
```
Memory footprint

Some limitations of the sparse matrix implementation in the *Matrix* package:

- nonzero values always stored as `double`, the most memory consuming type
- number of nonzero values must be $< 2^{31}$
- limited to 2 dimensions: no support for arrays of arbitrary number of dimensions
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
RleMatrix/RleArray and HDF5Matrix/HDF5Array provide:

- support all R atomic types
- no limits in size (but each dimension must be $< 2^{31}$)
- arbitrary number of dimensions

And also:

- delayed operations
- block processing (behind the scene)
- TODO: multicore block processing (sequential only at the moment)
Delayed operations

We start with HDF5Matrix object $M$:

```r
M <- as(m, "HDF5Matrix")
M
```

```r
## <63677 x 8> HDF5Matrix object of type "integer":
## [1,] 679 448 873 408 1138 1047 770 572
## [2,] 0   0   0   0   0   0   0   0
## [3,] 467 515 621 365 587 799 417 508
## [4,] 260 211 263 164 245 331 233 229
## [5,] 60  55  40  35  78  63  76  60
## ... . . . . . . . .
## [63673,] 0   0   0   1   0   1   0   0
## [63674,] 0   0   0   0   0   0   0   0
## [63675,] 0   0   0   0   0   0   0   0
## [63676,] 0   0   1   0   0   0   0   0
## [63677,] 0   0   0   0   1   0   0   0
```
RleArray and HDF5Array objects

Subsetting is delayed:

```r
M2 <- M[10:12, 1:5]
M2
## <3 x 5> DelayedMatrix object of type "integer":
## [1,] 394 236 464 175 658
## [2,] 172 168 264 118 241
## [3,] 2112 1867 5137 2657 2735
```

```r
seed(M2)
## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpnkzHDU/HDF5Array_dump/auto00002.h5"
## Slot "name":
## [1] "/HDF5ArrayAUTO00002"
## Slot "as_sparse":
## [1] FALSE
## Slot "type":
## [1] NA
## Slot "dim":
## [1] 63677 8
## Slot "chunkdim":
## [1] 63677 8
## Slot "first_val":
## [1] 679
```
RleArray and HDF5Array objects

Transposition is delayed:

```
M3 <- t(M2)
M3
```

```r
## <5 x 3> DelayedMatrix object of type "integer":
## [,1] [,2] [,3]
## [1,] 394  172 2112
## [2,] 236  168 1867
## [3,] 464  264 5137
## [4,] 175  118 2657
## [5,] 658  241 2735
```

```
seed(M3)
```

```r
## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpnkzHDU/HDF5Array_dump/auto00002.h5"
## Slot "name":
## [1] "/HDF5ArrayAUTO00002"
## Slot "as_sparse":
## [1] FALSE
## Slot "type":
## [1] NA
## Slot "dim":
## [1] 63677  8
## Slot "chunkdim":
## [1] 63677  8
## Slot "first_val":
## [1] 679
```
RleArray and HDF5Array objects

cbind() / rbind() are delayed:

```r
M4 <- cbind(M3, M[1:5, 6:8])
M4
## <5 x 6> DelayedMatrix object of type "integer":
## [1,]  394  172  2112  1047   770  572
## [2,]  236  168  1867   0     0     0
## [3,]  464  264  5137   799  417  508
## [4,]  175  118  2657  331  233  229
## [5,]  658  241  2735   63    76   60

t种子(M4)  # Error! (more than one seed)
```

Hervé Pages hpages.on.github@gmail.com (Bio conductor conference Boston)

Working with large arrays in R

July 2017 18 / 34
RleArray and HDF5Array objects

All the operations in the following groups are delayed:

- Arith (+, -, ...)
- Compare (==, <, ...)
- Logic (&, |)
- Math (log, sqrt)
- and more ...

```r
M5 <- M == 0
M5
```

```r
## <63677 x 8> DelayedMatrix object of type "logical":
## [1,] FALSE FALSE FALSE . FALSE FALSE
## [2,] TRUE TRUE TRUE . TRUE TRUE
## [3,] FALSE FALSE FALSE . FALSE FALSE
## [4,] FALSE FALSE FALSE . FALSE FALSE
## [5,] FALSE FALSE FALSE . FALSE FALSE
## ... . . . . . . . .
## [63673,] TRUE TRUE TRUE . TRUE TRUE
## [63674,] TRUE TRUE TRUE . TRUE TRUE
## [63675,] TRUE TRUE TRUE . TRUE TRUE
## [63676,] TRUE TRUE FALSE . TRUE TRUE
## [63677,] TRUE TRUE TRUE . TRUE TRUE
```

```r
seed(M5)
```

```r
## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpnkzHDU/HDF5Array_dump/auto00002.h5"
##
## Slot "name":
## [1] "/HDF5ArrayAUTO00002"
##
## Slot "as_sparse":
## [1] FALSE
##
## Slot "type":
## [1] NA
##
## Slot "dim":
## [1] 63677 8
##
## Slot "chunkdim":
## [1] 63677 8
##
## Slot "first_val":
## [1] 679
```
RleArray and HDF5Array objects

M6 <- round(M[11:14, ] / M[1:4, ], digits=3)
M6

## <4 x 8> DelayedMatrix object of type "double":
## [1,] 0.253 0.375 0.302 . 0.201 0.309
## [2,] Inf Inf Inf . Inf Inf
## [3,] 1.122 0.948 1.027 . 1.182 0.935
## [4,] 0.273 0.242 0.802 . 0.575 0.751

seed(M6)  # Error! (more than one seed)
RleArray and HDF5Array objects

Realization

Delayed operations can be **realized** by coercing the DelayedMatrix object to HDF5Array:

```r
M6a <- as(M6, "HDF5Array")
M6a
## <4 x 8> HDF5Matrix object of type "double":
## [1,] 0.253 0.375 0.302 . 0.201 0.309
## [2,] Inf Inf Inf . Inf Inf
## [3,] 1.122 0.948 1.027 . 1.182 0.935
## [4,] 0.273 0.242 0.802 . 0.575 0.751
```

```r
seed(M6a)
## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpnkzHDU/HDF5Array_dump/auto00003.h5"
## Slot "name":
## [1] "/HDF5ArrayAUTO00003"
## Slot "as_sparse":
## [1] FALSE
## Slot "type":
## [1] NA
## Slot "dim":
## [1] 4 8
## Slot "chunkdim":
## [1] 4 8
## Slot "first_val":
## [1] 0.253
```
... or by coercing it to RleArray:

```r
M6b <- as(M6, "RleArray")
M6b
```

```
## <4 x 8> RleMatrix object of type "double":
## [1,] 0.253 0.375 0.302 . 0.201 0.309
## [2,] Inf Inf Inf . Inf Inf
## [3,] 1.122 0.948 1.027 . 1.182 0.935
## [4,] 0.273 0.242 0.802 . 0.575 0.751
```

```
seed(M6b)
```

```
## An object of class "ChunkedRleArraySeed"
## Slot "breakpoints":
## [1] 32
## Slot "type":
## [1] "double"
## Slot "chunks":
## <environment: 0x563fb25e37f0>
## Slot "DIM":
## [1] 4 8
## Slot "DIMNAMES":
## [[1]]
## NULL
## [[2]]
## NULL
```
RleArray and HDF5Array objects

Controlling where HDF5 datasets are realized

**HDF5 dump management utilities**: a set of utilities to control where HDF5 datasets are written to disk.

```r
hdf5_dumpfile <- file.path(mydata_dir, "M6c.h5")
setHDF5DumpFile(hdf5_dumpfile)
setHDF5DumpName("M6c")
M6c <- as(M6, "HDF5Array")
seed(M6c)
## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpnkzHDU/mydata/M6c.h5"
## Slot "name":
## [1] "/M6c"
## Slot "as_sparse":
## [1] FALSE
## Slot "type":
## [1] NA
## Slot "dim":
## [1] 4 8
## Slot "chunkdim":
## [1] 4 8
## Slot "first_val":
## [1] 0.253
h5ls(hdf5_dumpfile)
## group name otype dclass dim
## 0 / M6c H5I_DATASET FLOAT 4 x 8
```
RleArray and HDF5Array objects

showHDF5DumpLog()

showHDF5DumpLog()

```r
## [2023-10-24 16:56:04.152725] #1 In file '/tmp/RtmpnkzHDU/HDF5Array_dump/auto00001.h5': creation of dataset '/HDF5ArrayAUTO00001' (63677x8:integer, chunkdims=63677x8, level=6)
## [2023-10-24 16:56:04.352194] #2 In file '/tmp/RtmpnkzHDU/HDF5Array_dump/auto00002.h5': creation of dataset '/HDF5ArrayAUTO00002' (63677x8:integer, chunkdims=63677x8, level=6)
## [2023-10-24 16:56:05.006517] #3 In file '/tmp/RtmpnkzHDU/HDF5Array_dump/auto00003.h5': creation of dataset '/HDF5ArrayAUTO00003' (4x8:double, chunkdims=4x8, level=6)
## [2023-10-24 16:56:05.232826] #4 In file '/tmp/RtmpnkzHDU/mydata/M6c.h5': creation of dataset 'M6c' (4x8:double, chunkdims=4x8, level=6)
```
Block processing

The following operations are NOT delayed. They are implemented via a *block processing* mechanism that loads and processes one block at a time:

- operations in the Summary group (`max`, `min`, `sum`, `any`, `all`)
- `mean`
- Matrix row/col summarization operations (`col/rowSums`, `col/rowMeans`, ...)
- `anyNA`, `which`
- `apply`
- and more ...
DelayedArray:::set_verbose_block_processing(TRUE)

## [1] FALSE

colSums(M)

## Processing block [[1/1, 1/1]] ... OK

## [1] 20637971 18809481 25348649 15163415 24448408 30818215 19126151 21164133

Control the block size:

getAutoBlockSize()

## [1] 1e+08

setAutoBlockSize(1e6)

## automatic block size set to 1e+06 bytes (was 1e+08)

colSums(M)

## Processing block [[1/1, 1/1]] ...

## [1] 20637971 18809481 25348649 15163415 24448408 30818215 19126151 21164133
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
Hands-on

1. Load the "airway" dataset.

2. It’s wrapped in a SummarizedExperiment object. Get the count data as an ordinary matrix.

3. Wrap it in an HDF5Matrix object: (1) using `writeHDF5Array()`; then (2) using coercion.

4. When using coercion, where has the data been written on disk?

5. See `?setHDF5DumpFile` for how to control the location of "automatic" HDF5 datasets. Try to control the destination of the data when coercing.
6. Use `showHDF5DumpLog()` to see all the HDF5 datasets written to disk during the current session.

7. Try some operations on the HDF5Matrix object: (1) some delayed ones; (2) some non-delayed ones (block processing).

8. Use `DelayedArray:::set_verbose_block_processing(TRUE)` to see block processing in action.

9. Control the block size with `setAutoBlockSize()`.
10. Stick the HDF5Matrix object back in the SummarizedExperiment object. The resulting object is an "HDF5-backed SummarizedExperiment object".

11. The HDF5-backed SummarizedExperiment object can be manipulated (almost) like an in-memory SummarizedExperiment object. Try [, cbind, rbind on it.

12. The SummarizedExperiment package provides saveHDF5SummarizedExperiment to save a SummarizedExperiment object (HDF5-backed or not) as an HDF5-backed SummarizedExperiment object. Try it.
Motivation and challenges

Memory footprint

RleArray and HDF5Array objects

Hands-on

DelayedArray/HDF5Array: Future developments
Future developments

Block processing improvements

Block genometry: (1) better by default, (2) let the user have more control on it

Support multicore

Expose it: blockApply()
Future developments

HDF5Array improvements

Store the dimnames in the HDF5 file (in *HDF5 Dimension Scale datasets* - https://www.hdfgroup.org/HDF5/Tutor/h5dimscale.html)

Use better automatic chunk geometry when realizing an HDF5Array object

Block processing should take advantage of the chunk geometry (e.g. `realize()` should use blocks that are clusters of chunks)

Unfortunately: not possible to support multicore realization at the moment (HDF5 does not support concurrent writing to a dataset yet)
**RleArray improvements**

Let the user have more control on the chunk geometry when constructing/realizing an RleArray object.

Like for HDF5Array objects, block processing should take advantage of the chunk geometry.

Support multicore realization.

Provide C/C++ low-level API for direct row/column access from C/C++ code (e.g. from the beachmat package).