Benchmarking the DelayedMatrixStats package

Peter Hickey

2017-06-04

Contents

Chapter 1

Introduction

The *DelayedArray* package defines the *DelayedMatrix* class, for wrapping matrix-like objects to provide a unified interface.

> Wrapping an matrix-like object (typically an on-disk object) in a DelayedMatrix object allows one to perform common matrix operations on it without loading the object in memory. In order to reduce memory usage and optimize performance, operations on the object are either delayed or executed using a block processing mechanism (see `?DelayedArray::DelayedMatrix` for further details)

A big advantage of this is we can use choose different matrix "*backends*" for storing the data while preserving a common interface. Examples of backends and data they are tailored towards using are given below:

Class	Package	Type of data	Example
matrix	*base*	Dense matrix	RNA-seq counts matrix
dgCMatrix	*Matrix*	Sparse data	Single-cell RNA-seq counts matrix
HDF5Matrix	*HDF5Array*	Dense matrix too large for memory	Non-CpG methylation
RleArray	*DelayedArray*	Data with runs of identical values	Sequencing coverage

1.1 Session info

The R session information when compiling this book is shown below:

```
devtools::session_info()
#>  setting  value
#>  version  R version 3.4.0 (2017-04-21)
#>  system   x86_64, darwin15.6.0
#>  ui       X11
#>  language (EN)
#>  collate  en_AU.UTF-8
#>  tz       America/New_York
#>  date     2017-06-04
#>
#>  package           * version    date
#>  backports           1.1.0      2017-05-22
```

```
#>  base                * 3.4.0      2017-04-21
#>  BiocGenerics        * 0.23.0     2017-04-27
#>  BiocStyle             2.5.1      2017-05-27
#>  bookdown              0.4        2017-06-04
#>  colorspace            1.3-2      2016-12-14
#>  compiler              3.4.0      2017-04-21
#>  datasets            * 3.4.0      2017-04-21
#>  DelayedArray        * 0.3.8      2017-06-02
#>  DelayedMatrixStats  * 0.0.0.9000 2017-05-28
#>  devtools              1.13.2     2017-06-02
#>  digest                0.6.12     2017-01-27
#>  evaluate              0.10       2016-10-11
#>  fortunes              1.5-4      2016-12-29
#>  ggplot2               2.2.1      2016-12-30
#>  graphics            * 3.4.0      2017-04-21
#>  grDevices           * 3.4.0      2017-04-21
#>  grid                  3.4.0      2017-04-21
#>  gtable                0.2.0      2016-02-26
#>  HDF5Array           * 1.5.8      2017-05-29
#>  htmltools             0.3.6      2017-04-28
#>  IRanges             * 2.11.3     2017-05-20
#>  knitr                 1.16       2017-05-18
#>  lattice               0.20-35    2017-03-25
#>  lazyeval              0.2.0      2016-06-12
#>  magrittr              1.5        2014-11-22
#>  Matrix              * 1.2-10     2017-04-28
#>  matrixStats         * 0.52.2     2017-04-14
#>  memoise               1.1.0      2017-05-26
#>  methods             * 3.4.0      2017-04-21
#>  microbenchmark      * 1.4-2.1    2015-11-25
#>  munsell               0.4.3      2016-02-13
#>  parallel            * 3.4.0      2017-04-21
#>  plyr                  1.8.4      2016-06-08
#>  profmem             * 0.4.0      2016-09-15
#>  Rcpp                  0.12.11    2017-05-22
#>  rhdf5               * 2.21.1     2017-05-11
#>  rlang                 0.1.1.9000 2017-05-26
#>  rmarkdown             1.5        2017-04-26
#>  rprojroot             1.2        2017-01-16
#>  rstudioapi            0.6        2016-06-27
#>  S4Vectors           * 0.15.2     2017-05-11
#>  scales                0.4.1      2016-11-09
#>  stats               * 3.4.0      2017-04-21
#>  stats4              * 3.4.0      2017-04-21
#>  stringi               1.1.5      2017-04-07
#>  stringr               1.2.0      2017-02-18
#>  tibble                1.3.3      2017-05-28
#>  tools                 3.4.0      2017-04-21
#>  utils               * 3.4.0      2017-04-21
#>  withr                 1.0.2      2016-06-20
#>  yaml                  2.1.14     2016-11-12
#>  zlibbioc              1.23.0     2017-04-27
#>  source
```

```
#>  CRAN (R 3.4.0)
#>  local
#>  Bioconductor
#>  Bioconductor
#>  Github (rstudio/bookdown@fdd68e4)
#>  CRAN (R 3.4.0)
#>  local
#>  local
#>  Bioconductor
#>  local
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  local
#>  local
#>  local
#>  CRAN (R 3.4.0)
#>  Bioconductor
#>  CRAN (R 3.4.0)
#>  Bioconductor
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  Github (hadley/memoise@e372cde)
#>  local
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  local
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  cran (@2.21.1)
#>  Github (hadley/rlang@c351186)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  cran (@0.15.2)
#>  CRAN (R 3.4.0)
#>  local
#>  local
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  cran (@1.3.3)
#>  local
#>  local
#>  CRAN (R 3.4.0)
#>  CRAN (R 3.4.0)
#>  Bioconductor
```

Chapter 2

Test data

We will benchmark the *DelayedMatrixStats* package using different types of matrix-like data:

- `dense_matrix`: A dense *matrix* with 600 columns and 20000 rows (91.6 Mb)
- `sparse_matrix`: A sparse *dgcMatrix* with 600 columns and 20000 rows where 60% of entries are zero (54.9 Mb)
- `rle_matrix`: An run-length encoded column *RleMatrix* with 6 columns and 2000000 rows (0.5 Mb)

```
library(DelayedMatrixStats)
library(Matrix)
library(HDF5Array)

# Dense matrix
dense_matrix <- DelayedArray(matrix(runif(20000 * 600), nrow = 20000,
                                    ncol = 600))
dense_matrix
#> DelayedMatrix object of 20000 x 600 doubles:
#>                [,1]       [,2]       [,3] ...     [,599]     [,600]
#>     [1,] 0.56211049 0.06052366 0.38033128   . 0.87799385 0.84693600
#>     [2,] 0.96372515 0.83906637 0.62496370   . 0.87406395 0.06190513
#>     [3,] 0.18319063 0.81058923 0.63978155   . 0.73985395 0.30726775
#>     [4,] 0.67745600 0.75232175 0.34283933   . 0.19136272 0.37044424
#>     [5,] 0.82650284 0.29491392 0.09360019   . 0.86290053 0.63351816
#>      ...          .          .          .   .          .          .
#> [19996,] 0.94517978 0.95109336 0.88648460   . 0.07201023 0.48243563
#> [19997,] 0.99308461 0.85953562 0.78748557   . 0.17937134 0.31886901
#> [19998,] 0.55040209 0.64421003 0.02668481   . 0.77054324 0.27607450
#> [19999,] 0.97592622 0.48989835 0.79181370   . 0.22407128 0.64161406
#> [20000,] 0.63496789 0.44949889 0.77485783   . 0.23457757 0.10117268

# 60% zero elements
sparse_matrix <- seed(dense_matrix)
zero_idx <- sample(length(sparse_matrix), 0.6 * length(sparse_matrix))
sparse_matrix[zero_idx] <- 0
sparse_matrix <- DelayedArray(Matrix::Matrix(sparse_matrix, sparse = TRUE))
sparse_matrix
#> DelayedMatrix object of 20000 x 600 doubles:
#>                [,1]       [,2]       [,3] ...     [,599]     [,600]
#>     [1,] 0.00000000 0.06052366 0.00000000   .  0.0000000  0.0000000
#>     [2,] 0.00000000 0.83906637 0.62496370   .  0.0000000  0.0000000
```

```
#>      [3,] 0.18319063 0.81058923 0.63978155   .  0.7398540  0.0000000
#>      [4,] 0.67745600 0.00000000 0.34283933   .  0.1913627  0.3704442
#>      [5,] 0.00000000 0.29491392 0.00000000   .  0.8629005  0.0000000
#>       ...          .          .          .   .          .          .
#> [19996,]  0.9451798  0.0000000  0.0000000   . 0.07201023 0.00000000
#> [19997,]  0.0000000  0.0000000  0.0000000   . 0.00000000 0.00000000
#> [19998,]  0.5504021  0.0000000  0.0000000   . 0.00000000 0.27607450
#> [19999,]  0.0000000  0.0000000  0.0000000   . 0.22407128 0.64161406
#> [20000,]  0.6349679  0.0000000  0.0000000   . 0.00000000 0.00000000

# HDF5-backed dense matrix
# hdf5_matrix <- as(dense_matrix, "HDF5Array")

# Run-length encoded column matrix
rle_matrix <- RleArray(Rle(sample(2L, 200000 * 6 / 10, replace = TRUE), 100),
                       dim = c(2000000, 6))
rle_matrix
#> RleMatrix object of 2000000 x 6 integers:
#>             [,1] [,2] [,3] [,4] [,5] [,6]
#>       [1,]    2    1    2    2    1    1
#>       [2,]    2    1    2    2    1    1
#>       [3,]    2    1    2    2    1    1
#>       [4,]    2    1    2    2    1    1
#>       [5,]    2    1    2    2    1    1
#>        ...    .    .    .    .    .    .
#> [1999996,]    1    1    1    1    1    2
#> [1999997,]    1    1    1    1    1    2
#> [1999998,]    1    1    1    1    1    2
#> [1999999,]    1    1    1    1    1    2
#> [2000000,]    1    1    1    1    1    2
```

Obviously, this is not a comprehensive set of inputs. Rather, it chosen to be somewhat representative of some typical genomics data.

2.1 Measuring performance

Timings are measured using the *microbenchmark* package using 10 repetitions (`times`). Memory allocations are reported using the *profmem* package.

```
library(microbenchmark)
library(profmem)
times <- 10
```

Chapter 3

colSums2()

We compare the `DelayedArray::colSums()` method, which uses the block-processing algorithm, to the `DelayedMatrixStats::colSums2()` method, which uses tailored algorithms depending on the *seed* of the *DelayedMatrix* and to the `DelayedMatrixStats:::.colSums2()` method that acts directly on the seed of the *DelayedMatrix* object[1]

3.1 Pristine *DelayedMatrix*

```
microbenchmark(DelayedArray::colSums(dense_matrix),
               DelayedMatrixStats::colSums2(dense_matrix),
               DelayedMatrixStats:::.colSums2(seed(dense_matrix)),
               times = times)
#> Unit: milliseconds
#>                                                expr        min         lq
#>                 DelayedArray::colSums(dense_matrix) 1237.05967 1286.72366
#>           DelayedMatrixStats::colSums2(dense_matrix)   13.75110   14.48974
#>  DelayedMatrixStats:::.colSums2(seed(dense_matrix))   12.24977   12.52132
#>        mean     median         uq        max neval
#>  1338.77309 1313.47916 1425.51354 1472.51304    10
#>    15.43412   14.89400   16.46142   19.15365    10
#>    13.46753   12.83785   13.45714   17.22766    10
total(profmem(DelayedArray::colSums(dense_matrix)))
#> [1] 2498171088
total(profmem(DelayedMatrixStats::colSums2(dense_matrix)))
#> [1] 165512

microbenchmark(DelayedArray::colSums(sparse_matrix),
               DelayedMatrixStats::colSums2(sparse_matrix),
               DelayedMatrixStats:::.colSums2(seed(sparse_matrix)),
               Matrix::colSums(seed(sparse_matrix)),
               times = times)
#> Unit: milliseconds
#>                                                 expr         min
#>                 DelayedArray::colSums(sparse_matrix) 1263.948627
#>           DelayedMatrixStats::colSums2(sparse_matrix)   11.125717
```

[1] `DelayedMatrixStats:::.colSums2()` shouldn't be called by the user because it does not realise delayed operations. It is used here for demonstration purposes on a "*pristine*" DelayedMatrix to measure the additional overhead of S4 methods

```
#>  DelayedMatrixStats:::.colSums2(seed(sparse_matrix))    9.694688
#>                 Matrix::colSums(seed(sparse_matrix))    9.831111
#>           lq       mean     median         uq        max neval
#>  1277.792760 1354.97051 1345.26165 1409.33254 1478.01487    10
#>    11.321978   13.61341   12.38058   14.26526   24.21124    10
#>     9.737193   11.27165   10.42035   11.44131   15.96811    10
#>    10.428975   11.36380   10.85824   12.76856   13.00290    10
total(profmem(DelayedArray::colSums(sparse_matrix)))
#> [1] 1709267496
total(profmem(DelayedMatrixStats::colSums2(sparse_matrix)))
#> [1] 5464

microbenchmark(DelayedArray::colSums(rle_matrix),
               DelayedMatrixStats::colSums2(rle_matrix),
               DelayedMatrixStats:::.colSums2(seed(rle_matrix)),
               times = times)
#> Unit: milliseconds
#>                                           expr         min          lq
#>                DelayedArray::colSums(rle_matrix) 2480.267737 2522.151174
#>          DelayedMatrixStats::colSums2(rle_matrix)    3.846588    4.031420
#>  DelayedMatrixStats:::.colSums2(seed(rle_matrix))    2.660000    2.794059
#>         mean      median          uq         max neval
#>  2564.381064 2555.504718 2616.171735 2646.783855    10
#>     4.632065    4.457076    4.814034    6.303599    10
#>     7.772226    3.021242    3.113427   50.727535    10

total(profmem(DelayedArray::colSums(rle_matrix)))
#> [1] 787403608
total(profmem(DelayedMatrixStats::colSums2(rle_matrix)))
#> [1] 1872
```

3.2 With row subsetting

```
i <- sample(nrow(dense_matrix), nrow(dense_matrix) / 10)
microbenchmark(DelayedArray::colSums(dense_matrix[i, ]),
               DelayedMatrixStats::colSums2(dense_matrix, rows = i),
               times = times)
#> Unit: milliseconds
#>                                              expr       min        lq
#>               DelayedArray::colSums(dense_matrix[i, ]) 150.12124 177.85944
#>  DelayedMatrixStats::colSums2(dense_matrix, rows = i)  14.09315  14.59356
#>       mean    median        uq       max neval
#>  198.11233 192.73521 226.85481 240.19939    10
#>   15.12774  14.92626  15.38786  17.04865    10
total(profmem(DelayedArray::colSums(dense_matrix[i, ])))
#> [1] 326614368
total(profmem(DelayedMatrixStats::colSums2(dense_matrix, rows = i)))
#> [1] 21512

microbenchmark(DelayedArray::colSums(sparse_matrix[i, ]),
               DelayedMatrixStats::colSums2(sparse_matrix, rows = i),
```

```
               times = times)
#> Unit: milliseconds
#>                                                  expr      min       lq
#>              DelayedArray::colSums(sparse_matrix[i, ]) 181.0150 185.5543
#>  DelayedMatrixStats::colSums2(sparse_matrix, rows = i)  43.4571  46.1228
#>       mean    median        uq       max neval
#>  241.06562 214.46952 260.04889 403.65730    10
#>   47.23405  47.07576  48.33854  51.88142    10
total(profmem(DelayedArray::colSums(sparse_matrix[i, ])))
#> [1] 217297184
total(profmem(DelayedMatrixStats::colSums2(sparse_matrix, rows = i)))
#> [1] 5805384

i <- sample(nrow(rle_matrix), nrow(rle_matrix) / 10)
microbenchmark(DelayedArray::colSums(rle_matrix[i, ]),
               DelayedMatrixStats::colSums2(rle_matrix, rows = i),
               times = times)
#> Unit: milliseconds
#>                                               expr       min       lq
#>              DelayedArray::colSums(rle_matrix[i, ])  266.4322  282.946
#>  DelayedMatrixStats::colSums2(rle_matrix, rows = i) 2509.9470 2536.812
#>       mean    median        uq       max neval
#>   288.4523  285.5148  287.8931  342.1545    10
#>  2565.2083 2557.8006 2567.9954 2645.7166    10
total(profmem(DelayedArray::colSums(rle_matrix[i, ])))
#> [1] 86579640
total(profmem(DelayedMatrixStats::colSums2(rle_matrix, rows = i)))
#> [1] 99980704
```

3.3 With column subsetting

```
j <- sample(ncol(dense_matrix), ncol(dense_matrix) / 10)
microbenchmark(DelayedArray::colSums(dense_matrix[, j]),
               DelayedMatrixStats::colSums2(dense_matrix, cols = j),
               times = times)
#> Unit: milliseconds
#>                                                 expr        min
#>              DelayedArray::colSums(dense_matrix[, j]) 130.822413
#>  DelayedMatrixStats::colSums2(dense_matrix, cols = j)   2.954383
#>          lq       mean     median         uq        max neval
#>  185.368641 192.741377 199.560679 204.350156 217.700374    10
#>    3.031592   3.534314   3.402916   3.717595   5.269981    10
total(profmem(DelayedArray::colSums(dense_matrix[, j])))
#> [1] 326835928
total(profmem(DelayedMatrixStats::colSums2(dense_matrix, cols = j)))
#> [1] 161192

microbenchmark(DelayedArray::colSums(sparse_matrix[, j]),
               DelayedMatrixStats::colSums2(sparse_matrix, cols = j),
               times = times)
#> Unit: milliseconds
```

```
#>                                                 expr       min        lq
#>              DelayedArray::colSums(sparse_matrix[, j]) 145.41955 151.62413
#>  DelayedMatrixStats::colSums2(sparse_matrix, cols = j)  12.24748  12.81168
#>       mean    median        uq      max neval
#>  187.63815 182.46718 222.64284 240.3814    10
#>   25.01388  14.35228  14.91043 124.2795    10
total(profmem(DelayedArray::colSums(sparse_matrix[, j])))
#> [1] 217156704
total(profmem(DelayedMatrixStats::colSums2(sparse_matrix, cols = j)))
#> [1] 5755072

j <- sample(ncol(rle_matrix), ncol(rle_matrix) / 2)
microbenchmark(DelayedArray::colSums(rle_matrix[, j]),
               DelayedMatrixStats::colSums2(rle_matrix, cols = j),
               times = times)
#> Unit: milliseconds
#>                                               expr         min         lq
#>              DelayedArray::colSums(rle_matrix[, j]) 1204.439061 1265.87484
#>  DelayedMatrixStats::colSums2(rle_matrix, cols = j)    3.645216    3.79055
#>        mean      median         uq         max neval
#>  1280.29217 1287.497136 1296.26175 1336.682742    10
#>     4.20517    4.107648    4.52946    4.896633    10
total(profmem(DelayedArray::colSums(rle_matrix[, j])))
#> [1] 401705608
total(profmem(DelayedMatrixStats::colSums2(rle_matrix, cols = j)))
#> [1] 1872
```

3.4 With row and column subsetting

```
i <- sample(nrow(dense_matrix), nrow(dense_matrix) / 10)
j <- sample(ncol(dense_matrix), ncol(dense_matrix) / 10)
microbenchmark(DelayedArray::colSums(dense_matrix[i, j]),
               DelayedMatrixStats::colSums2(dense_matrix, rows = i, cols = j),
               times = times)
#> Unit: milliseconds
#>                                                      expr       min
#>                      DelayedArray::colSums(dense_matrix[i, j]) 38.678242
#>  DelayedMatrixStats::colSums2(dense_matrix, rows = i, cols = j)  2.471122
#>         lq      mean    median        uq       max neval
#>  40.658299 63.969289 67.446417 71.177195 103.697163    10
#>   2.514982  2.757317  2.719349  2.876589   3.446033    10
microbenchmark(DelayedArray::colSums(sparse_matrix[i, j]),
               DelayedMatrixStats::colSums2(sparse_matrix, rows = i, cols = j),
               times = times)
#> Unit: milliseconds
#>                                                       expr       min
#>                      DelayedArray::colSums(sparse_matrix[i, j]) 58.16497
#>  DelayedMatrixStats::colSums2(sparse_matrix, rows = i, cols = j) 44.19810
#>        lq     mean   median       uq      max neval
#>  61.23579 72.97175 64.44056 68.37965 156.5290    10
#>  45.33201 47.71227 47.05405 50.12441  52.1516    10
```

```
i <- sample(nrow(rle_matrix), nrow(rle_matrix) / 10)
j <- sample(ncol(rle_matrix), ncol(rle_matrix) / 2)
microbenchmark(DelayedArray::colSums(rle_matrix[i, j]),
               DelayedMatrixStats::colSums2(rle_matrix, rows = i, cols = j),
               times = times)
#> Unit: milliseconds
#>                                                        expr       min
#>                        DelayedArray::colSums(rle_matrix[i, j])  144.6618
#>  DelayedMatrixStats::colSums2(rle_matrix, rows = i, cols = j) 1227.4698
#>         lq      mean   median       uq      max neval
#>   146.4575  150.6922  147.420  148.844  179.0782    10
#>  1245.4921 1263.5119 1253.124 1269.115 1353.0914    10
```

3.5 With delayed ops

```
microbenchmark(DelayedArray::colSums(dense_matrix ^ 2),
               DelayedMatrixStats::colSums2(dense_matrix ^ 2),
               times = times)
#> Unit: milliseconds
#>                                         expr       min        lq
#>         DelayedArray::colSums(dense_matrix^2) 1326.8490 1431.4468
#>  DelayedMatrixStats::colSums2(dense_matrix^2)  246.0587  266.2838
#>       mean    median       uq      max neval
#>  1484.4255 1496.4808 1545.328 1591.2768    10
#>   302.2183  299.9772  310.943  413.7981    10
microbenchmark(DelayedArray::colSums(sparse_matrix ^ 2),
               DelayedMatrixStats::colSums2(sparse_matrix ^ 2),
               times = times)
#> Unit: milliseconds
#>                                         expr       min        lq
#>         DelayedArray::colSums(sparse_matrix^2) 1368.0792 1402.4556
#>  DelayedMatrixStats::colSums2(sparse_matrix^2)  531.5231  567.5002
#>       mean    median       uq      max neval
#>  1486.7550 1466.0952 1545.2155 1652.8075    10
#>   582.7944  587.3182  602.8364  621.8481    10
microbenchmark(DelayedArray::colSums(rle_matrix ^ 2),
               DelayedMatrixStats::colSums2(rle_matrix ^ 2),
               times = times)
#> Unit: seconds
#>                                       expr      min       lq     mean
#>         DelayedArray::colSums(rle_matrix^2) 2.617288 2.673016 2.765875
#>  DelayedMatrixStats::colSums2(rle_matrix^2) 2.330117 2.380350 2.441969
#>    median       uq      max neval
#>  2.715906 2.887769 2.987700    10
#>  2.418112 2.530877 2.547293    10
```

3.6 Summary

Bibliography