Benchmarking the DelayedMatrixStats package

Peter Hickey

2017-06-09

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     2017-06-09
#>
#>  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       * 0.3.9      2017-06-03
#>  DelayedMatrixStats * 0.0.0.9000 2017-06-09
#>  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          * 0.15.3     2017-06-03
#>  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)
#>  Bioconductor
#>  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,] 0.291123149 0.746567243 0.714732255   . 0.75857413 0.64889705
#>     [2,] 0.695032771 0.835652019 0.994773283   . 0.57136535 0.59089245
#>     [3,] 0.376537710 0.076370680 0.631883153   . 0.08632942 0.64604683
#>     [4,] 0.099155580 0.983260972 0.033870884   . 0.17194848 0.78613336
#>     [5,] 0.223655179 0.003016347 0.184510716   . 0.32769976 0.98399021
#>      ...           .           .           .   .          .          .
#> [19996,]  0.19562940  0.64344791  0.34820057   .  0.2382142  0.5758769
#> [19997,]  0.50863128  0.38698444  0.34155233   .  0.5391943  0.2431121
#> [19998,]  0.85569166  0.31043662  0.78594727   .  0.7177261  0.8931934
#> [19999,]  0.90838796  0.04155858  0.36675316   .  0.5686322  0.3579163
#> [20000,]  0.21878846  0.74873281  0.99661168   .  0.7821497  0.5828003

# 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,] 0.000000000 0.000000000 0.000000000   . 0.7585741 0.6488971
#>     [2,] 0.695032771 0.835652019 0.000000000   . 0.0000000 0.0000000
```

```
#>      [3,] 0.376537710 0.076370680 0.631883153   . 0.0000000 0.6460468
#>      [4,] 0.099155580 0.983260972 0.000000000   . 0.0000000 0.0000000
#>      [5,] 0.223655179 0.003016347 0.000000000   . 0.0000000 0.0000000
#>       ...           .           .           .   .         .         .
#> [19996,]  0.19562940  0.64344791  0.00000000   . 0.0000000 0.0000000
#> [19997,]  0.00000000  0.00000000  0.34155233   . 0.0000000 0.2431121
#> [19998,]  0.00000000  0.31043662  0.00000000   . 0.7177261 0.8931934
#> [19999,]  0.00000000  0.04155858  0.36675316   . 0.5686322 0.0000000
#> [20000,]  0.00000000  0.00000000  0.99661168   . 0.0000000 0.5828003

# 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    2    2    2    1    1
#>       [2,]    2    2    2    2    1    1
#>       [3,]    2    2    2    2    1    1
#>       [4,]    2    2    2    2    1    1
#>       [5,]    2    2    2    2    1    1
#>        ...    .    .    .    .    .    .
#> [1999996,]    1    2    1    2    2    2
#> [1999997,]    1    2    1    2    2    2
#> [1999998,]    1    2    1    2    2    2
#> [1999999,]    1    2    1    2    2    2
#> [2000000,]    1    2    1    2    2    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*, as well as comparing to the `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)),
               matrixStats::colSums2(seed(dense_matrix)),
               times = times)
#> Unit: milliseconds
#>                                              expr        min         lq
#>               DelayedArray::colSums(dense_matrix) 1283.48911 1864.31577
#>        DelayedMatrixStats::colSums2(dense_matrix)   17.29489   21.24989
#>  DelayedMatrixStats::colSums2(seed(dense_matrix))   13.60255   17.87302
#>         matrixStats::colSums2(seed(dense_matrix))   12.99146   15.80239
#>        mean     median         uq        max neval
#>  2004.46189 2048.45751 2301.36485 2521.58140    10
#>    38.08371   36.05451   47.70144   73.72097    10
#>    25.94223   24.00078   30.68420   41.72278    10
#>    21.81281   17.31656   28.85252   33.48672    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)
```

[1]The 'seed method' 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

```
#> Unit: milliseconds
#>                                                 expr         min          lq
#>                DelayedArray::colSums(sparse_matrix) 1264.530744 1316.15718
#>         DelayedMatrixStats::colSums2(sparse_matrix)   11.817381   13.42313
#>  DelayedMatrixStats::colSums2(seed(sparse_matrix))   10.175122   11.20796
#>                 Matrix::colSums(seed(sparse_matrix))    9.694093   11.23716
#>        mean     median          uq         max neval
#>  1452.48860 1386.43823 1524.39434 1853.55755    10
#>    14.41638   13.99009   15.73583   17.47814    10
#>    13.53717   12.41742   14.05605   21.09214    10
#>    11.71826   11.74576   12.26451   13.80694    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) 1310.023207 1355.152398
#>         DelayedMatrixStats::colSums2(rle_matrix)    4.044929    4.127730
#>  DelayedMatrixStats::colSums2(seed(rle_matrix))    2.654707    2.824271
#>         mean      median          uq         max neval
#>  1406.699029 1417.552379 1430.568801 1561.240497    10
#>     6.835946    4.509377    8.610080   15.512422    10
#>     3.166907    3.067424    3.324333    4.297344    10

total(profmem(DelayedArray::colSums(rle_matrix)))
#> [1] 594934472
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, ]) 232.66977 269.30946
#>  DelayedMatrixStats::colSums2(dense_matrix, rows = i)  17.61365  17.94966
#>       mean    median        uq         max neval
#>  398.3456 322.16767 402.06056 1075.46715    10
#>   23.7913  23.64738  28.12027   31.48741    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, ]) 180.50731 216.85644
#>  DelayedMatrixStats::colSums2(sparse_matrix, rows = i)  45.46621  51.43771
#>       mean    median        uq      max neval
#>  308.51542 254.18478 440.75724 513.4355    10
#>   72.10438  53.29957  70.54412 148.0876    10
total(profmem(DelayedArray::colSums(sparse_matrix[i, ])))
#> [1] 217293464
total(profmem(DelayedMatrixStats::colSums2(sparse_matrix, rows = i)))
#> [1] 5801656

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, ])  134.5513  145.7055
#>  DelayedMatrixStats::colSums2(rle_matrix, rows = i) 2532.1597 2576.5126
#>       mean    median        uq       max neval
#>   175.0424  152.0624  193.1471  315.7631    10
#>  2791.6364 2636.5898 2961.2963 3686.7874    10
total(profmem(DelayedArray::colSums(rle_matrix[i, ])))
#> [1] 62582200
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]) 158.172784
#>  DelayedMatrixStats::colSums2(dense_matrix, cols = j)   3.507745
#>          lq      mean     median         uq        max neval
#>  160.532558 182.67404 163.364146 164.916947 314.841775    10
#>    3.833441   4.50892   4.257297   4.693838   6.984677    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]) 157.9222 183.6137
#>  DelayedMatrixStats::colSums2(sparse_matrix, cols = j)  11.3190  13.4754
#>       mean    median        uq       max neval
#>  213.04790 193.51740 216.82503 330.93421    10
#>   14.90289  15.52494  16.24069  17.78912    10
total(profmem(DelayedArray::colSums(sparse_matrix[, j])))
#> [1] 217168584
total(profmem(DelayedMatrixStats::colSums2(sparse_matrix, cols = j)))
#> [1] 5766944

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]) 674.01695 692.114868
#>  DelayedMatrixStats::colSums2(rle_matrix, cols = j)   3.81455   4.432296
#>        mean     median         uq        max neval
#>  737.488282 733.975966 766.304464 823.837522    10
#>    4.888522   4.565458   4.685504   8.740521    10
total(profmem(DelayedArray::colSums(rle_matrix[, j])))
#> [1] 305470160
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]) 36.368492
#>  DelayedMatrixStats::colSums2(dense_matrix, rows = i, cols = j)  2.998825
#>         lq      mean    median        uq       max neval
#>  39.064818 58.958453 64.690798 71.119834 85.491278    10
#>   3.029176  3.559399  3.125859  3.900654  5.987884    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]) 46.43427
```

```
#>  DelayedMatrixStats::colSums2(sparse_matrix, rows = i, cols = j) 43.84288
#>        lq     mean   median       uq       max neval
#>  58.80461 69.18963 64.13646 67.67738 131.87153    10
#>  45.14225 47.68512 47.19202 50.18447  52.70835    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])   74.0634
#>  DelayedMatrixStats::colSums2(rle_matrix, rows = i, cols = j) 1258.5250
#>          lq      mean     median        uq       max neval
#>    80.39394  103.6879   96.71862  117.3074  153.2567    10
#>  1278.73703 1582.9624 1391.34098 1916.1114 2196.0357    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) 1807.3580 1965.1023
#>  DelayedMatrixStats::colSums2(dense_matrix^2)  351.0731  404.5719
#>       mean    median        uq       max neval
#>  2055.0077 2003.8251 2178.5791 2290.5040    10
#>   519.8276  507.9676  582.1483  864.0489    10
microbenchmark(DelayedArray::colSums(sparse_matrix ^ 2),
               DelayedMatrixStats::colSums2(sparse_matrix ^ 2),
               times = times)
#> Unit: milliseconds
#>                                          expr      min       lq     mean
#>         DelayedArray::colSums(sparse_matrix^2) 1829.675 1922.903 2159.961
#>  DelayedMatrixStats::colSums2(sparse_matrix^2)  682.596  727.075  796.014
#>     median        uq       max neval
#>  2099.8463 2200.3313 2890.7803    10
#>   792.6587  851.5654  989.4052    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) 1.621425 1.658672 1.735662
#>  DelayedMatrixStats::colSums2(rle_matrix^2) 1.303282 1.369719 1.460526
#>    median       uq      max neval
#>  1.720114 1.768661 2.009433    10
#>  1.433386 1.553991 1.632721    10
```

3.6 Summary

Bibliography