Lecture 14 - I/O and Out-of-Core Methods

DSE 512

Drew Schmidt
2022-03-10

From Last Time

- Homework not graded
- New homework "soon"
- Questions?

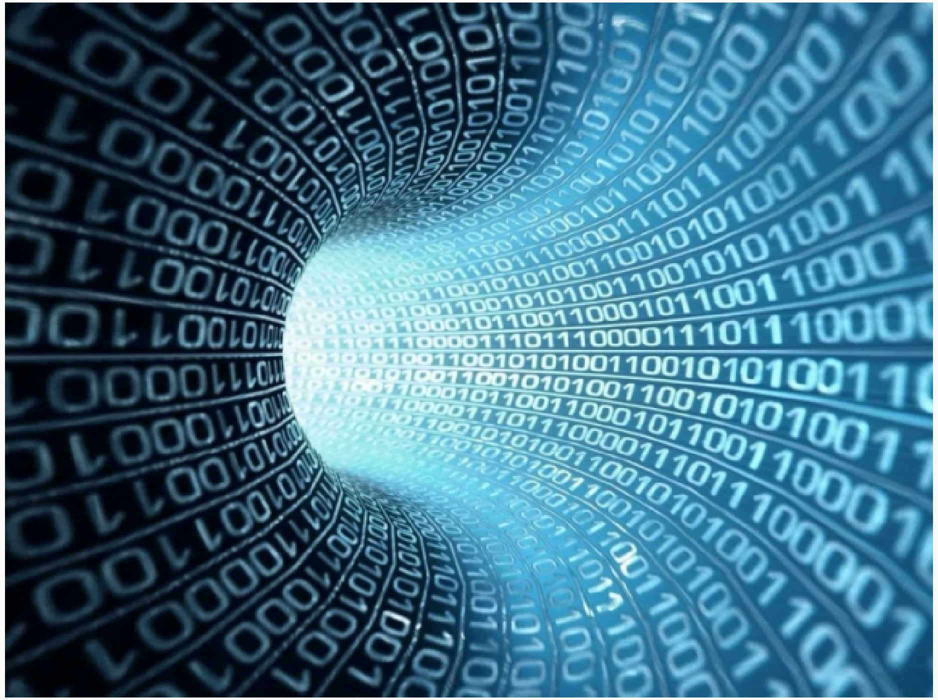
Data Larger Than Memory?

Four choices:

1. ~~Give up!~~
2. ~~Downsample~~
3. Go out of core
4. Go parallel (distributed)

I/O

- Hard to get wrong on a laptop
- Hard to get right on a cluster
- People get PhD's in I/O!



Standards

- Industry/the cloud examples
  - binary: databases, binary xml, custom formats
  - text: csv, log files, ...
- HPC
  - binary: HDF5, netcdf, custom formats
  - text: csv, log files, ...
- Other
  - language serialization (pickle, rds files)

Binary File Formats: SQL

- Quasi-standard
- Many implementations (postgres, sqlite, ...)
- Unusual inside academia; UBIQUITOUS in industry

Binary File Formats: HDF5

- Hierarchical Data Format
- Multi-dimensional array serialization
- *Many* features; *very* complicated

HDF5 Bindings

- Python
  - h5py https://www.h5py.org/
  - pandas!
- R
  - low level
    - rhdf5 https://www.bioconductor.org/packages/release/bioc/html/rhdf5.html
    - hdf5r https://cran.r-project.org/web/packages/hdf5r/index.html
  - high level
    - hdfio https://github.com/RBigData/hdfio
    - hdfmat https://hpcran.org/packages/hdfmat/index.html
- The interface greatly determines the usage

h5py

Write

```
import numpy as np
import h5py

np.random.seed(1234)
x = np.random.random(size=(1000,20))

hw = h5py.File("/tmp/data.h5", "w")
hw.create_dataset("mydataset", data=x)
```

```
<HDF5 dataset "mydataset": shape (1000, 20), type "<f8">
```

```
hw.close()
```

Read

```
hr = h5py.File("/tmp/data.h5", "r")
hr.keys()
```

```
<KeysViewHDF5 ['mydataset']>
```

```
hr["mydataset"][:]
hr.close()
```

Write

```r
library(hdf5r)

set.seed(1234)
x = matrix(runif(1000*20), 1000, 20)

hw = H5File$new("/tmp/data.h5", mode = "w
hw[["mydataset"]] = x
hw
```

```
Class: H5File
Filename: /tmp/data.h5
Access type: H5F_ACC_RDWR
Listing:
      name    obj_type dataset.dims dataset.type_class
 mydataset H5I_DATASET    1000 x 20          H5T_FLOAT
```

```r
hw$close_all()
```

Read

```r
hr = H5File$new("/tmp/data.h5", mode = "r
hr[["mydataset"]][,]
hw$close_all()
```

hdfmat

```r
library(hdfmat)
set.seed(1234)

f = tempfile()
m = 20
n = 1000
x = matrix(rnorm(m*n), m, n)
h = hdfmat::hdfmat(f, "mydata", n, n, "fl
h$fill_crossprod(x)
```

```r
h
```

```
# An hdfmat object
  * Location: /tmp/Rtmpdf6Akd/file213ddf45859855
  * Dimension: 1000x1000
  * Type: float
```

```r
h$read(row_start=1, row_stop=1)
```

```
# A float32 matrix: 1x1000
       [,1]    [,2]   [,3]   [,4]   [,5]
[1,] 20.785 -1.3239 -3.432 4.3694 2.9657
# ...
```

```r
h$eigen(k=3)
```

```
# A float32 vector: 3
[1] 1.1637e+03 8.4397e+02 8.5777e-03
```

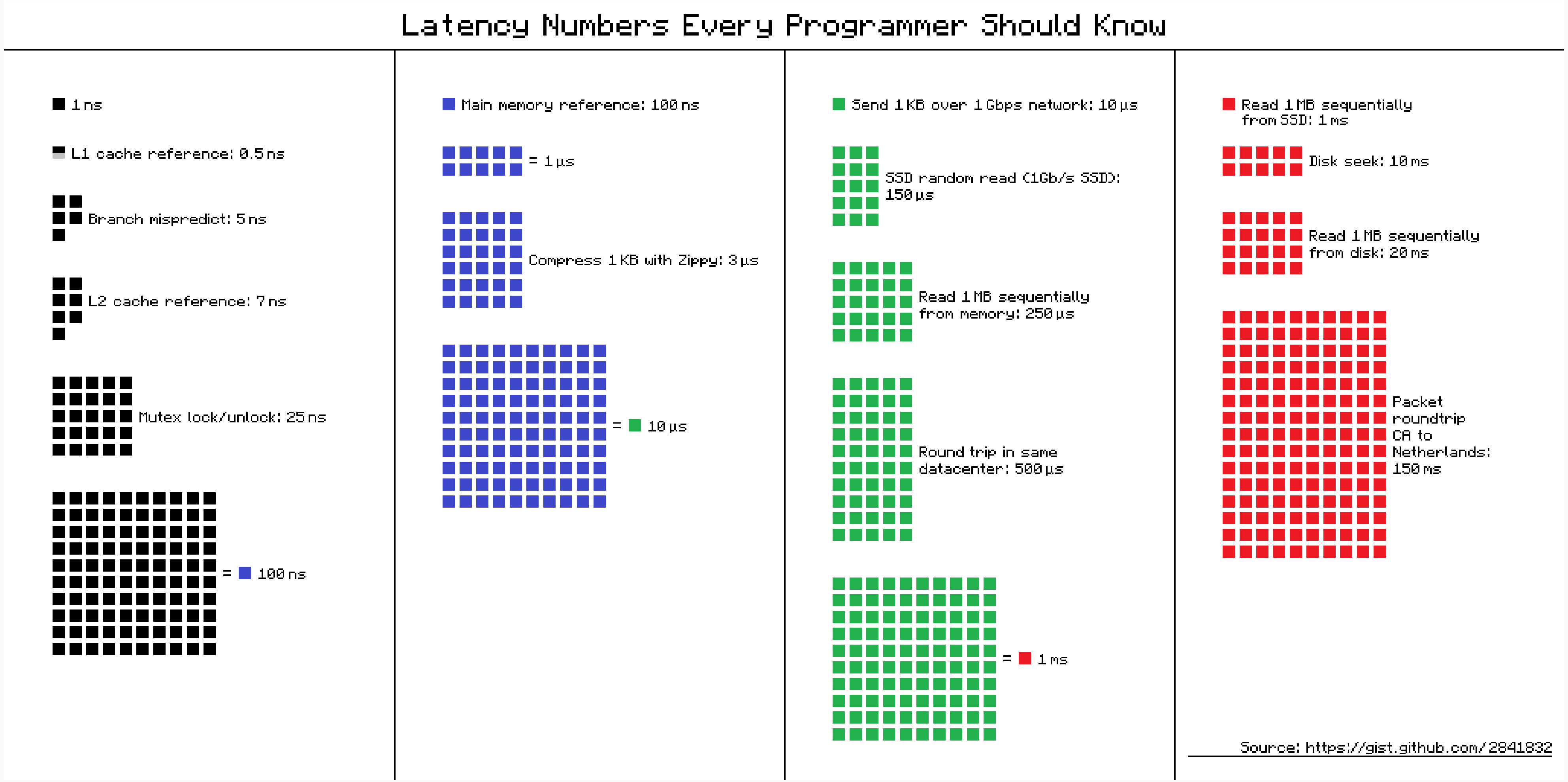
Out-of-Core

Out-of-Core Processing

- Data held externally to main memory (i.e., on the file system)
- Process data in "blocks" or "chunks"
  - Read block
  - Process block
  - Continue
- Ideal data storage for out-of-core work
  - binary
  - sub-setting not unreasonably difficult

THIS IS NOT FAST

Speed Comparisons



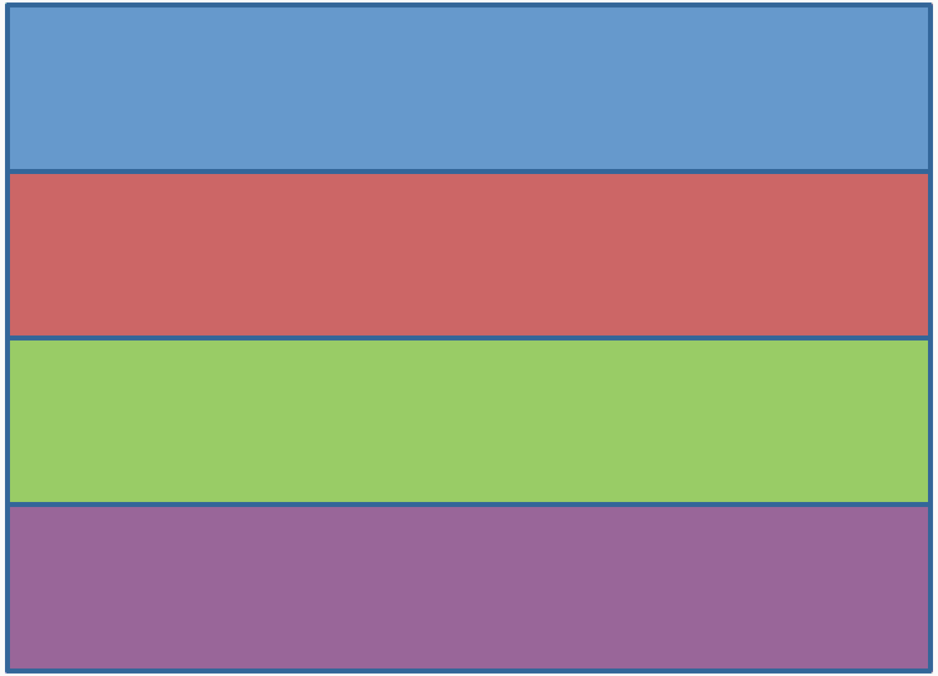
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Resources

- Relative Memory Access Speeds https://www.overbyte.com.au/misc/Lesson3/CacheFun.html
- What Every Programmer Should Know About Memory https://www.gwern.net/docs/cs/2007-drepper.pdf
- The Pathologies of Big Data https://queue.acm.org/detail.cfm?id=1563874

Out-of-Core Strategies

- Blocks of rows
  - Operate on collections of contiguous rows
  - "1-d" distribution
- Blocks of columns
- 2-d block cyclic distributions



We will only look at "blocks of rows".

Two Sets of Issues

1. Algorithm modification
2. Index juggling

Algorithm Modification

- Obviously, situation specific
- Do as much as you can with the data you have before moving on
- Very similar to parallel programming

Index Juggling

- Need to know which rows to read
- Split m rows into b blocks
- Free choices
  - 0 or 1-based?
  - Include lower boundary?
  - Include upper boundary?

We will do 1-based, both boundaries included.

Index Juggling Helpers

- Choose number of rows within each block (except possibly the last)
- Determines the number of blocks (i.e., number of iterations)

```
get_num_blocks = function(num_rows, rows_per_block){
  ceiling(num_rows / rows_per_block)
}
```

Index Juggling Helpers

- Low - Index of the within-block first row
- High - Index of of the within-block last row

```
blockid_to_indices = function(blockid, num_rows, rows_per_block)
{
  ind_low = (blockid-1)*rows_per_block + 1
  ind_high = min(num_rows, ind_low + rows_per_block - 1)
  list(ind_low = ind_low, ind_high = ind_high)
}
```

Strategies Can Get Convoluted

Choosing params based on memory consumption

```
memuse::Sys.meminfo()
```

```
## Totalram:  62.810 GiB
## Freeram:   44.219 GiB
```

```
num_cols = 500
pct_of_ram <- 0.15
ram_per_block <- pct_of_ram * memuse::Sys.meminfo()$totalram
rows_per_block <- memuse::howmany(ram_per_block, ncol=num_cols)[1]
rows_per_block
```

```
## [1] 2529058
```

Example: SQL

```r
library(DBI)
library(RSQLite)

# create fake data
set.seed(1234)
n = 100
big_tbl = data.frame(
  ind = 1:n,
  x = runif(n),
  y = rnorm(n)
)

# write table to disk
db = dbConnect(RSQLite::SQLite(), "/tmp/d
dbWriteTable(db, "big_tbl", big_tbl)
```

```r
# Process out-of-core
rpb = 7
num_blocks = get_num_blocks(n, rpb)
for (blockid in 1:num_blocks){
  ids = blockid_to_indices(blockid, n, rp
  query = paste(
    "SELECT * FROM big_tbl WHERE ind >=",
    ids$ind_low,
    "AND ind <=",
    ids$ind_high
  )
  sub_tbl = dbGetQuery(db, query)
  print(sub_tbl)
}

# close connection
dbDisconnect(db)
```

Example: SQL

First few

```
  ind           x          y
1   1 0.113703411 -1.8060313
2   2 0.622299405 -0.5820759
3   3 0.609274733 -1.1088896
4   4 0.623379442 -1.0149620
5   5 0.860915384 -0.1623095
6   6 0.640310605  0.5630558
7   7 0.009495756  1.6478175
  ind         x           y
1   8 0.2325505 -0.77335342
2   9 0.6660838  1.60590963
3  10 0.5142511 -1.15780855
4  11 0.6935913  0.65658846
5  12 0.5449748  2.54899107
6  13 0.2827336 -0.03476039
7  14 0.9234335 -0.66963358
```

Last few

```
  ind         x          y
1  92 0.9004246  1.3951479
2  93 0.1340782  0.6366744
3  94 0.1316141 -0.1084317
4  95 0.1052875  0.5137628
5  96 0.5115836  0.3992718
6  97 0.3001991  1.6628564
7  98 0.0267169  0.2758934
  ind         x         y
1  99 0.3096474 0.5062726
2 100 0.7421197 0.3475520
```

Example: hdf5

```r
library(hdf5r)

# create fake data
set.seed(1234)
n = 100
big_tbl = data.frame(
  x = runif(n),
  y = rnorm(n)
)

# write table to disk
hw = H5File$new("/tmp/data.h5", mode = "w'
hw[["big_tbl"]] = as.matrix(big_tbl)
hw$close_all()
```

```r
# Process out-of-core
hr = H5File$new("/tmp/data.h5", mode = "r'

rpb = 7
num_blocks = get_num_blocks(n, rpb)
for (blockid in 1:num_blocks){
  ids = blockid_to_indices(blockid, n, rpk
  sub_tbl = hr[["big_tbl"]][ids$ind_low:ic
  print(sub_tbl)
}

# close connection
hw$close_all()
```

Example: hdf5

First few

```
              [,1]       [,2]
[1,] 0.113703411 -1.8060313
[2,] 0.622299405 -0.5820759
[3,] 0.609274733 -1.1088896
[4,] 0.623379442 -1.0149620
[5,] 0.860915384 -0.1623095
[6,] 0.640310605  0.5630558
[7,] 0.009495756  1.6478175
            [,1]        [,2]
[1,] 0.2325505 -0.77335342
[2,] 0.6660838  1.60590963
[3,] 0.5142511 -1.15780855
[4,] 0.6935913  0.65658846
[5,] 0.5449748  2.54899107
[6,] 0.2827336 -0.03476039
[7,] 0.9234335 -0.66963358
```

Last few

```
            [,1]       [,2]
[1,] 0.9004246  1.3951479
[2,] 0.1340782  0.6366744
[3,] 0.1316141 -0.1084317
[4,] 0.1052875  0.5137628
[5,] 0.5115836  0.3992718
[6,] 0.3001991  1.6628564
[7,] 0.0267169  0.2758934
            [,1]      [,2]
[1,] 0.3096474 0.5062726
[2,] 0.7421197 0.3475520
```

Some Performance Considerations

- Open your connection ***ONCE***
- Block sizes
  - Memory/performance tradeoffs
  - Larger is better for runtime
  - Best performance: everything in memory
- Consider your file format
  - hdf5 is row-major
  - R is column major
  - *this implies a transpose in memory*

Out-of-Core SVD

Connection to Eigendecomposition

$$
\begin{aligned}
A^T A &= \left(U\Sigma V^T\right)^T \left(U\Sigma V^T\right) \\
&= V\Sigma U^T U\Sigma V^T \\
&= V\Sigma^2 V^T
\end{aligned}
$$

Computing the "Normal Equations" Matrix

Choose $b > 0$ and split A into b blocks of rows:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_b \end{bmatrix}$$

Then

$$A^T A = \sum_{i=1}^{b} A_i^T A_i$$

Example

```r
set.seed(1234)
m = 1000
n = 3
A = matrix(rnorm(m*n), m, n)
A_1 = A[1:300, ]
A_2 = A[301:899, ]
A_3 = A[900:1000, ]
```

```r
crossprod(A)
```

```
##             [,1]      [,2]       [,3]
## [1,] 994.39527  54.97576   14.77759
## [2,]  54.97576 961.98697  -34.08700
## [3,]  14.77759 -34.08700 1024.64272
```

```r
crossprod(A_1) + crossprod(A_2) + crosspr
```

```
##             [,1]      [,2]       [,3]
## [1,] 994.39527  54.97576   14.77759
## [2,]  54.97576 961.98697  -34.08700
## [3,]  14.77759 -34.08700 1024.64272
```

Out-of-Core SVD Algorithm

- Inputs
  - $A_{m \times n}$
  - Number of blocks b
- Procedure
  - Initialize $B_{n \times n} = 0$
  - For each $1 \leq i \leq b$
    - Read block of rows A_i
    - Compute $B = B + A_i^T A_i$
  - Factor $B = \Lambda \Delta \Lambda$
- Return $A = U \Sigma V^T$ where
  - $\Sigma = \sqrt{\Delta}$
  - $V = \Lambda$
  - $U = AV\Sigma^{-1}$

Helpers

```
get_num_blocks = function(num_rows, rows_per_block){
  ceiling(num_rows / rows_per_block)
}

blockid_to_indices = function(blockid, num_rows, rows_per_block)
{
  ind_low = (blockid-1)*rows_per_block + 1
  ind_high = min(num_rows, ind_low + rows_per_block - 1)
  list(ind_low = ind_low, ind_high = ind_high)
}
```

Example: SVD with SQL

```
svd(big_tbl[, -1])$v
```

```
##              [,1]       [,2]
## [1,] 0.0495752  0.9987704
## [2,] 0.9987704 -0.0495752
```

```
B = matrix(0, 2, 2)
rpb = 7
num_blocks = get_num_blocks(n, rpb)
for (blockid in 1:num_blocks){
  ids = blockid_to_indices(blockid, n, rpb
  query = paste(
    "SELECT * FROM big_tbl WHERE ind >=",
    ids$ind_low,
    "AND ind <=",
    ids$ind_high
  )
  A_i = dbGetQuery(db, query)
  B = B + crossprod(as.matrix(A_i[, -1]))
}
svd(B)$vt
```

```
##              [,1]       [,2]
## [1,] 0.0495752  0.9987704
## [2,] 0.9987704 -0.0495752
```

PCA?

- Have SVD
- Need mean-center-er
- 2-pass method:
  - For each blockid:
    - Read row block
    - Compute running total column sums
  - Divide column sums by number of rows
  - For each blockid:
    - Read row block
    - Mean-center columns
    - Write row block

hdfmat Case Study

- Background
  - SVD via Lanczos Iteration https://fml-fam.github.io/blog/2020/06/15/svd-via-lanczos-iteration/
  - Matrix Computations in Constrained Memory Environments https://fml-fam.github.io/blog/2021/06/29/matrix-computations-in-constrained-memory-environments/
- Computing Eigenvalues Out-of-Core https://fml-fam.github.io/blog/2021/10/25/computing-eigenvalues-out-of-core/

Out-of-Core Regression

Regression

- Normal equations
- QR
- SVD
- Solving the optimization problem

Regression: Optimization Problem

$$\min_{\beta \in \mathbb{R}^n} \frac{1}{2m} \sum_{i=1}^{m} \left((X\beta)_i - y_i\right)^2$$

```
cost_gaussian = function(beta, x, y){
  m = nrow(x)
  (1/(2*m))*sum((x%*%beta - y)^2)
}

reg.fit = function(x, y, maxiter=100){
  control = list(maxit=maxiter)
  beta = numeric(ncol(x))
  optim(par=beta, fn=cost_gaussian, x=x, y=y, method="CG", control=control)
}
```

```
reg.fit(X, y)$par
```

Setup

```r
library(hdf5r)
set.seed(1234)

f = "/tmp/data.h5"
m = 1000
n = 20
x = matrix(runif(m*n), m, n)
y = runif(m)

hw = H5File$new(f, mode = "w")
hw[["x"]] = x
hw[["y"]] = y
hw$close_all()
```

Helpers

```r
get_num_blocks = function(num_rows, rows_per_block){
  ceiling(num_rows / rows_per_block)
}

blockid_to_indices = function(blockid, num_rows, rows_per_block)
{
  ind_low = (blockid-1)*rows_per_block + 1
  ind_high = min(num_rows, ind_low + rows_per_block - 1)
  list(ind_low = ind_low, ind_high = ind_high)
}
```

Regression with hdf5

$$\min_{\beta \in \mathbb{R}^n} \frac{1}{2m} \sum_{i=1}^{m} \left((X\beta)_i - y_i\right)^2$$

```
cost_gaussian = function(beta, h, rpb){
  m = h[["x"]]$dims[1]
  num_blocks = get_num_blocks(m, rpb)

  s = 0
  for (blockid in 1:num_blocks){
    ids = blockid_to_indices(blockid, m,
    x = h[["x"]][ids$ind_low:ids$ind_high
    y = h[["y"]][ids$ind_low:ids$ind_high
    s = s + sum((x%*%beta - y)^2)
  }

  (1/(2*m))*s
}
```

```
reg.fit = function(h5_file, rpb, maxiter=
  hr = H5File$new(h5_file, mode = "r")

  control = list(maxit=maxiter)
  beta = numeric(ncol(x))
  ret = optim(par=beta, h=hr, rpb=rpb,
    fn=cost_gaussian,
    method="CG", control=control
  )

  hr$close_all()
  ret
}
```

Demonstration

```
reg.fit(f, rpb=17)$par
```

```
[1] 0.06566188 0.07943443 0.04653829 0.05206409 0.04032249 0.05792940
[7] 0.05956420 0.06516913 0.02962817 0.05392435 0.06939655 0.02545636
[13] 0.02184218 0.01626945 0.06479390 0.04442636 0.06176416 0.01739670
[19] 0.06109669 0.05337342
```

```
lm.fit(x, y)$coef
```

```
        x1         x2         x3         x4         x5         x6         x7 
0.06566687 0.07943613 0.04654031 0.05207004 0.04032480 0.05793326 0.05957063 
        x8         x9        x10        x11        x12        x13        x14 
0.06517145 0.02962701 0.05392347 0.06939197 0.02545156 0.02183941 0.01626128 
       x15        x16        x17        x18        x19        x20 
0.06479252 0.04441006 0.06176820 0.01740154 0.06109558 0.05337311 
```

Timings

```
system.time(reg.fit(f, rpb=100)$par)[3]
```

```
elapsed
 98.923
```

```
system.time(reg.fit(f, rpb=1000)$par)[3]
```

```
elapsed
 21.315
```

Questions?