Big Data Analytics

Lecture 2: Computation and Memory

Prof. Dr. Ulrich Matter

04/03/2021

Updates

Suggested Learning Procedure

- Get familiar with Git/GitHub (create an account if you do not have one yet).
- Clone/fork the course’s GitHub-repository.
- During class, use the Rmd-file of the slide-set as basis for your notes.
- After class, enrich/merge/extend your notes with the lecture notes.

Group Examinations

- Make sure to build teams of 3 (exception: 2).
- Register your team (and all members) here.
- **Team name**: important for handing out team assignments via GitHub Classroom.

Recap Week 1

Focus in This Course

- How to use the existing machinery most efficiently for large amounts of data?
- How to approach the analysis of large amounts of data with econometrics?
  1. Compute ‘usual’ statistics based on large dataset (many observations).
  2. Practical handling of large data sets for applied econometrics (gathering, storage, preparation, etc.)

R used in two ways

- A tool to analyze problems posed by large datasets.
  - For example, memory usage (in R).
- A practical tool for Big Data Analytics.

Introductory Example

Naïve approach (ignorant of R)

```r
# Naïve approach (ignorant of R)
deflator <- 1.05 # define deflator
# iterate through each observation
pce_real <- list()
n_obs <- length(economics$pce)
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real <- c(pce_real, economics$pce[i]/deflator)
})

time_elapsed
```

```
##    user  system elapsed
##   0.091   0.004   0.095
```

Introductory Example

Consider memory allocation

```r
# Improve memory allocation (still somewhat ignorant of R)
deflator <- 1.05 # define deflator
n_obs <- length(economics$pce)
pce_real <- list()
# allocate memory beforehand
# tell R how long the list will be
length(pce_real) <- n_obs
# iterate through each observation
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real[[i]] <- economics$pce[i]/deflator
})

time_elapsed
```

```
##    user  system elapsed 
##   0.005   0.000   0.005
```

Introductory Example

Fastest implementation (vectorization)

```r
library(microbenchmark)
# measure elapsed time in microseconds (avg.)
time_elapsed <-
  summary(microbenchmark(pce_real <- economics$pce/deflator))$mean

# per row (in sec)
time_per_row <- (time_elapsed/n_obs)/10^6
```

Introductory Example

Fastest implementation (vectorization)

```
# in seconds
(time_per_row*100^4)
```

```
## [1] 0.1354656
```

```
# in minutes
(time_per_row*100^4)/60
```

```
## [1] 0.00225776
```

```
# in hours
(time_per_row*100^4)/60^2
```

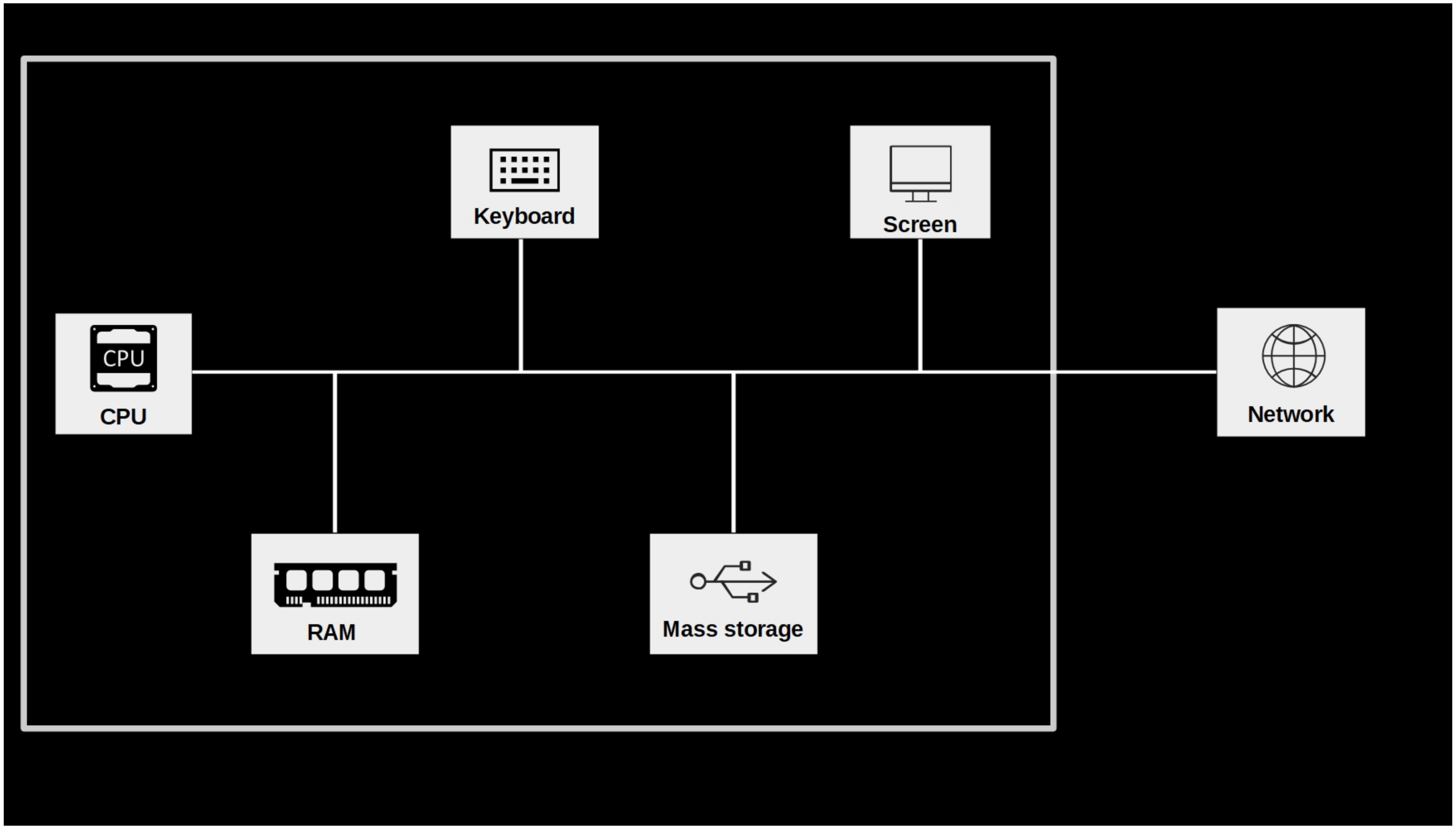
```
## [1] 3.762933e-05
```

Computation and Memory

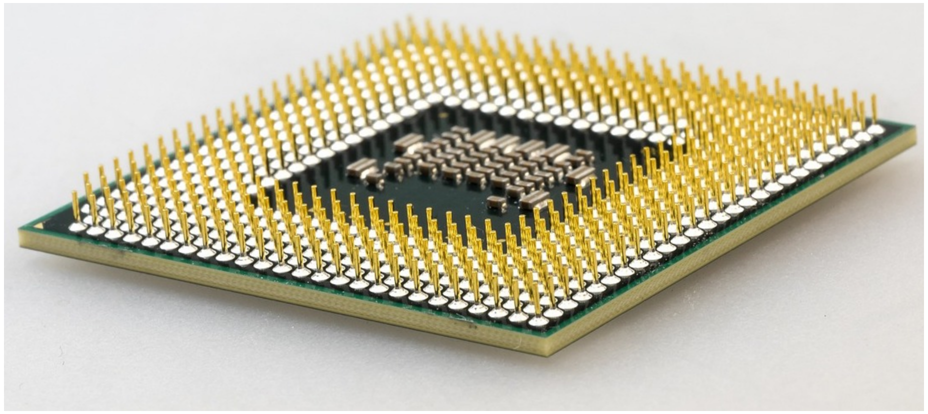
Goals of Today

- Know the core components of a computer
- Understand the very basics of information processing on a computer
- Have a basic idea of how to think about and approach computational challenges
  - From a statistics/applied econometrics perspective
  - From a computational/programmatic perspective
- Repetition of econometrics concepts: OLS, Monte Carlo, clustered SEs

Components of a standard computing environment

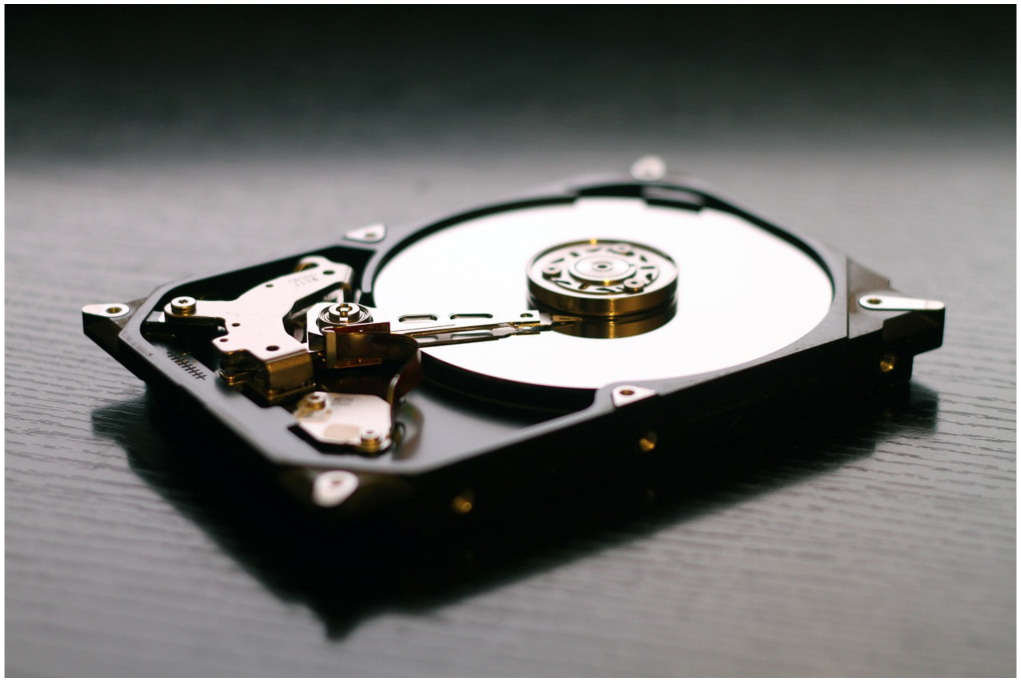


Central Processing Unit



Random Access Memory

Mass storage: hard drive



Now, what is Big Data (Analytics)?

- **Big Data Analytics**: The amount of data to be analyzed is not compatible with the standard usage of one or several of the computing environment’s hardware components (the components fail or work very inefficiently).

Now, what is Big Data (Analytics)?

- **Big Data Analytics**: The amount of data to be analyzed is not compatible with the standard usage of one or several of the computing environment's hardware components (the components fail or work very inefficiently).
- Need to understand how to **make best use of the available resources**, given a specific data analysis task.

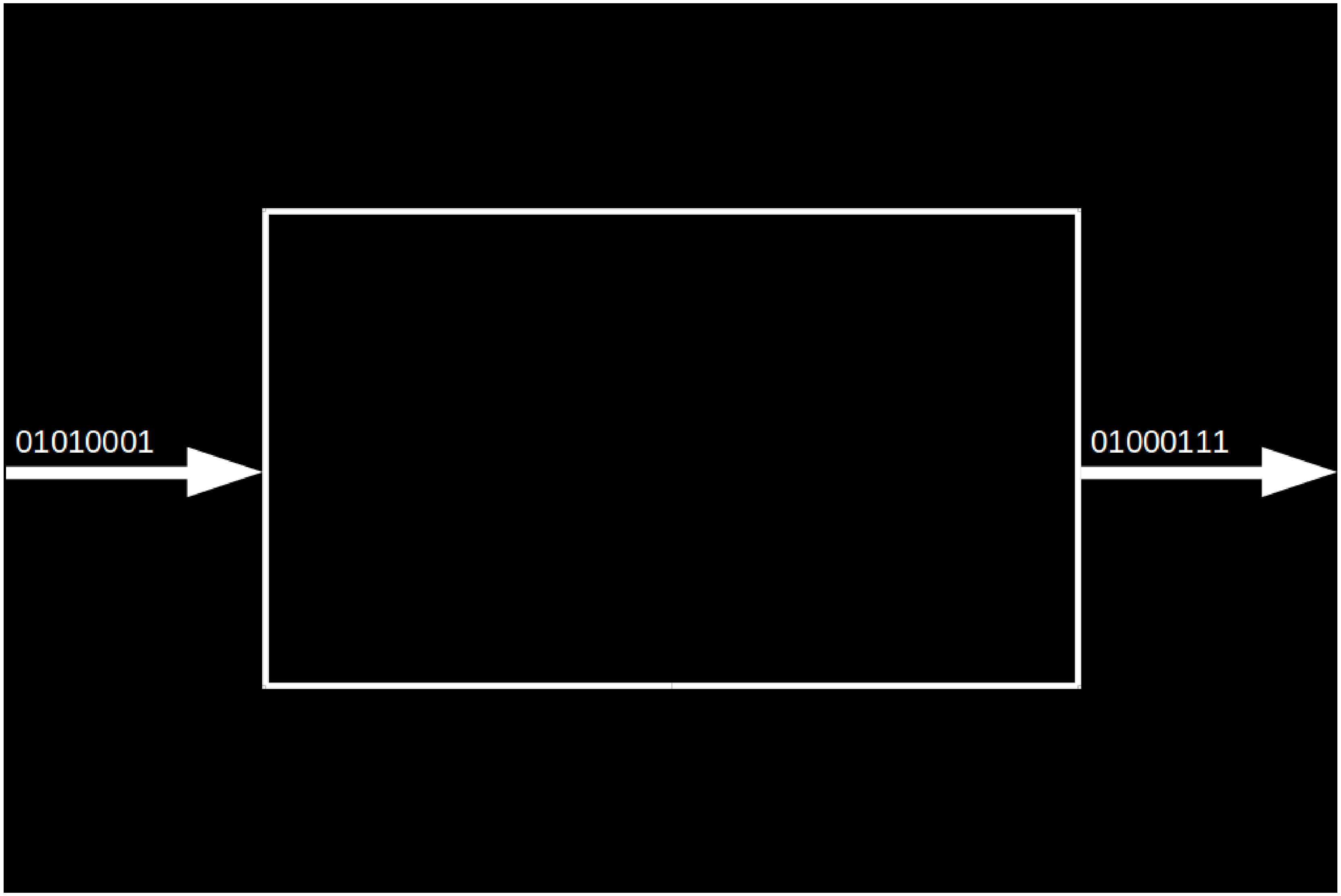
Now, what is Big Data (Analytics)?

- Big Data Analytics: The amount of data to be analyzed is not compatible with the standard usage of one or several of the computing environment’s hardware components (the components fail or work very inefficiently).
- Need to understand how to make best use of the available resources, given a specific data analysis task.
  - CPU: Parallel processing (use all cores available)
  - RAM: Efficient memory allocation and usage
  - RAM + Mass Storage: Virtual memory, efficient swapping

Now, what is Big Data (Analytics)?

- **Big Data Analytics**: The amount of data to be analyzed is not compatible with the standard usage of one or several of the computing environment's hardware components (the components fail or work very inefficiently).
- Need to understand how to **make best use of the available resources**, given a specific data analysis task.
    - CPU: Parallel processing (use all cores available)
    - RAM: Efficient memory allocation and usage
    - RAM + Mass Storage: Virtual memory, efficient swapping
- Make use of alternative/faster **statistical procedures**

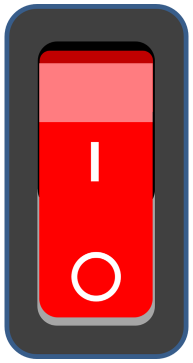
Units of information/data storage


01010001
01000111

The binary system

Microprocessors can only represent two signs (states):

- ‘Off’ = `0`
- ‘On’ = `1`



The binary system

- Only two signs: `0`, `1`.
- Base 2.
- Columns: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, and so forth.

The binary system

What is the decimal number 139 in the binary counting frame?

The binary system

What is the decimal number 139 in the binary counting frame?

- Solution:

$$(1 \times 2^7) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

The binary system

What is the decimal number 139 in the binary counting frame?

- Solution:

$$(1 \times 2^7) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

- More precisely:

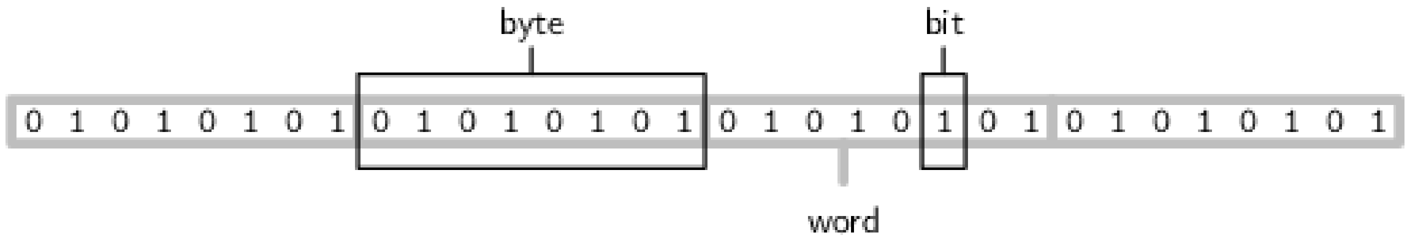
$$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) \\ + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

- That is, the number `139` in the decimal system corresponds to `10001011` in the binary system.

Units of information storage

- Smallest unit (a `0` or a `1`): **bit** (from **bi**nary dig**it**; abbrev. ‘b’).
- **Byte** (1 byte = 8 bits; abbrev. ‘B’)
  - For example, `10001011` (`139`)
- (4 bytes (or 32 bits) are called a **word**.)

Units of information storage



Bit, Byte, Word. Figure by Murrell (2009) (licensed under [CC BY-NC-SA 3.0 NZ](CC BY-NC-SA 3.0 NZ))

Units of information storage

Bigger units for storage capacity usually build on bytes:

- 1 kilobyte (KB) $= 1000^1 \approx 2^{10}$ bytes
- 1 megabyte (MB) $= 1000^2 \approx 2^{20}$ bytes
- 1 gigabyte (GB) $= 1000^3 \approx 2^{30}$ bytes
- 1 terabyte (TB) $= 1000^4 \approx 2^{40}$ bytes
- 1 petabyte (PB) $= 1000^5 \approx 2^{50}$ bytes
- 1 exabyte (EB) $= 1000^6 \approx 2^{60}$ bytes
- 1 zettabyte (ZB) $= 1000^7 \approx 2^{70}$ bytes

$$1ZB = 1000000000000000000000 \text{ bytes} = 1 \text{ billion terabytes.}$$

Information storage and data types

- Binary code can be interpreted in different ways
- As text, number, etc.
- Depending on the data type of a string of symbols such as `139`, more or less bytes are needed to represent it.

Example Size of R Objects

```r
object.size("139")
```

```
## 112 bytes
```

```r
object.size(139)
```

```
## 56 bytes
```

Computation in Applied Econometrics

Computational burden

- How to identify bottle-necks?
- Two aspects
    - How is a statistic computed?
    - How does the program/software/language implement this computation?

Two ways of approaching the burden

1. Consider alternative statistical procedures that happen to be more efficient (here: computationally efficient in contrast to statistically efficient). To compute a given statistic based on large amounts of data.
2. Given the statistical procedure, how to implement it **efficiently** in your computing environment?

What if this is not enough?

- Already using all components most efficiently?
- Scale up (‘vertical scaling’)
- Scale out (‘horizontal scaling’)

Faster Statistical Procedures

Example: Fast Least Squares Regression

- Classical approach to estimating linear models: OLS.
- Alternative: The **Uluru** algorithm (Dhillon et al. 2013).

OLS as a point of reference

Recall the OLS estimator in matrix notation, given the linear model $\mathbf{y} = \mathbf{X}\beta + \epsilon$:

$$\hat{\beta}_{OLS} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}.$$

Computational bottleneck of OLS

- $\hat{\beta}_{OLS}$ depends on $(\mathbf{X}^\intercal\mathbf{X})^{-1}$.
- Large cross-product if the number of observations is large (X is of dimensions $n \times p$)
- (Large) matrix inversions are computationally demanding.
- OLS has a $O(np^2)$ running time.

OLS in R

```r
beta_ols <-
     function(X, y) {

          # compute cross products and inverse
          XXi <- solve(crossprod(X,X))
          Xy <- crossprod(X, y)

          return( XXi  %*% Xy )
     }
```

Monte Carlo study

- Parameters and pseudo data

```r
# set parameter values
n <- 10000000
p <- 4

# Generate sample based on Monte Carlo
# generate a design matrix (~ our 'dataset') with four variables and 10000 observations
X <- matrix(rnorm(n*p, mean = 10), ncol = p)
# add column for intercept
X <- cbind(rep(1, n), X)
```

Monte Carlo study

- Model and model output

```
# MC model
y <- 2 + 1.5*X[,2] + 4*X[,3] - 3.5*X[,4] + 0.5*X[,5] + rnorm(n)
```

Monte Carlo study

- Performance of OLS

```
# apply the ols estimator
beta_ols(X, y)
```

```
##            [,1]
## [1,]  1.9966035
## [2,]  1.5001925
## [3,]  3.9999142
## [4,] -3.4998863
## [5,]  0.5001352
```

The Uluru algorithm as an alternative to OLS

Following Dhillon et al. (2013), we compute $\hat{\beta}_{Uluru}$:

$$\hat{\beta}_{Uluru} = \hat{\beta}_{FS} + \hat{\beta}_{correct}$$

, where

$$\hat{\beta}_{FS} = (\mathbf{X}_{subs}^{\intercal}\mathbf{X}_{subs})^{-1}\mathbf{X}_{subs}^{\intercal}\mathbf{y}_{subs}$$

, and

$$\hat{\beta}_{correct} = \frac{n_{subs}}{n_{rem}} \cdot (\mathbf{X}_{subs}^{\intercal}\mathbf{X}_{subs})^{-1}\mathbf{X}_{rem}^{\intercal}\mathbf{R}_{rem}$$

, and

$$\mathbf{R}_{rem} = \mathbf{Y}_{rem} - \mathbf{X}_{rem} \cdot \hat{\beta}_{FS}$$

.

The Uluru algorithm as an alternative to OLS

- Key idea: Compute $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ only on a sub-sample (X_{subs}, etc.)
- If the sample is large enough (which is the case in a Big Data context), the result is approximately the same.

Uluru algorithm in R (simplified)

```r
# simple version of the Uluru algorithm
beta_uluru <-
     function(X_subs, y_subs, X_rem, y_rem) {

          # compute beta_fs (this is simply OLS applied to the subsample)
          XXi_subs <- solve(crossprod(X_subs, X_subs))
          Xy_subs <- crossprod(X_subs, y_subs)
          b_fs <- XXi_subs  %*% Xy_subs

          # compute \mathbf{R}_{rem}
          R_rem <- y_rem - X_rem %*% b_fs

          # compute \hat{\beta}_{correct}
          b_correct <- (nrow(X_subs)/(nrow(X_rem))) * XXi_subs %*% crossprod(X_rem, R_rem)

          # beta uluru
          return(b_fs + b_correct)
     }
```

Uluru algorithm in R (simplified)

Test it with the same input as above:

```r
# set size of subsample
n_subs <- 1000
# select subsample and remainder
n_obs <- nrow(X)
X_subs <- X[1L:n_subs,]
y_subs <- y[1L:n_subs]
X_rem <- X[(n_subs+1L):n_obs,]
y_rem <- y[(n_subs+1L):n_obs]

# apply the uluru estimator
beta_uluru(X_subs, y_subs, X_rem, y_rem)
```

```
##            [,1]
## [1,]  1.9634975
## [2,]  1.5016897
## [3,]  3.9974782
## [4,] -3.4992864
## [5,]  0.5034944
```

Uluru algorithm: Monte Carlo study

```r
# define subsamples
n_subs_sizes <- seq(from = 1000, to = 500000, by=10000)
n_runs <- length(n_subs_sizes)
# compute uluru result, stop time
mc_results <- rep(NA, n_runs)
mc_times <- rep(NA, n_runs)
for (i in 1:n_runs) {
     # set size of subsample
     n_subs <- n_subs_sizes[i]
     # select subsample and remainder
     n_obs <- nrow(X)
     X_subs <- X[1L:n_subs,]
     y_subs <- y[1L:n_subs]
     X_rem <- X[(n_subs+1L):n_obs,]
     y_rem <- y[(n_subs+1L):n_obs]

     mc_results[i] <- beta_uluru(X_subs, y_subs, X_rem, y_rem)[2] # the first element is the interce
     mc_times[i] <- system.time(beta_uluru(X_subs, y_subs, X_rem, y_rem))[3]

}
```

Uluru algorithm: Monte Carlo study

```r
# compute ols results and ols time
ols_time <- system.time(beta_ols(X, y))
ols_res <- beta_ols(X, y)[2]
```

Uluru algorithm: Monte Carlo study

- Visualize comparison with OLS.

```
# load packages
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     economics
```

```
# prepare data to plot
plotdata <- data.frame(beta1 = mc_results,
                       time_elapsed = mc_times,
                       subs_size = n_subs_sizes)
```

Uluru algorithm: Monte Carlo study

1. Computation time.

```
ggplot(plotdata, aes(x = subs_size, y = time_elapsed)) +
     geom_point(color="darkgreen") +
     geom_hline(yintercept = ols_time[3],
                color = "red",
                size = 1) +
     theme_minimal() +
     ylab("Time elapsed") +
     xlab("Subsample size")
```

Uluru algorithm: Monte Carlo study

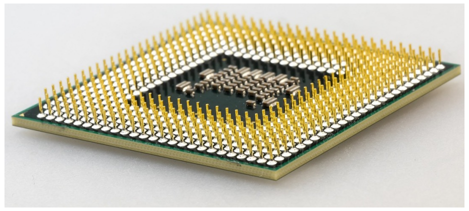
1. Precision

```r
ggplot(plotdata, aes(x = subs_size, y = beta1)) +
     geom_hline(yintercept = ols_res,
                color = "red",
                size = 1) +
     geom_hline(yintercept = 1.5,
                color = "green",
                size = 1) +
     geom_point(color="darkgreen") +
     theme_minimal() +
     ylab("Estimated coefficient") +
     xlab("Subsample size")
```

Efficient Use of Resources

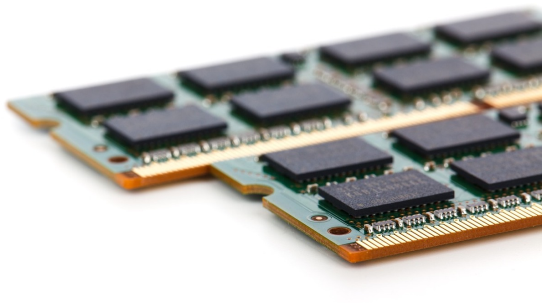
1) Parallel processing: CPU/core

- A CPU on any modern computer has several **cores**.
- The OS usually assigns automatically which tasks/processes should run on which core.
- We can explicitly instruct the computer to dedicate N cores to a specific computational task: **parallel processing**.



2) Memory allocation: RAM

- Standard computation procedures happen **in-memory**: data needs to be loaded into RAM.
- Default lower-level procedures to **allocate memory** might not be optimal for large data sets.
- We can explicitly use **faster** memory allocation procedures for a specific big data task.



3) Beyond RAM: virtual memory

- What if we run out of RAM?
- The OS deals with this by using part of the hard disk as **virtual memory**.
- By explicitly instructing the computer how to use **virtual memory for specific big data tasks**, we can speed things up.

Case study: Parallel processing

We start with importing the data into R.

```r
url <- "https://vincentarelbundock.github.io/Rdatasets/csv/carData/MplsStops.csv"
stopdata <- data.table::fread(url) # skipNul avoids running into encoding issues with this data set
```

Case study: Parallel processing

First, let's remove observations with missing entries (`NA`) and code our main explanatory variable and the dependent variable.

```r
# remove incomplete obs
stopdata <- na.omit(stopdata)
# code dependent var
stopdata$vsearch <- 0
stopdata$vsearch[stopdata$vehicleSearch=="YES"] <- 1
# code explanatory var
stopdata$white <- 0
stopdata$white[stopdata$race=="White"] <- 1
```

Case study: Parallel processing

We specify our baseline model as follows.

```r
model <- vsearch ~ white + factor(policePrecinct)
```

Case study: Parallel processing

And estimate the linear probability model via OLS (the `lm` function).

```
fit <- lm(model, stopdata)
summary(fit)
```

```
##
## Call:
## lm(formula = model, data = stopdata)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.13937 -0.06329 -0.05473 -0.04227  0.97729
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)              0.054733   0.005154  10.619  < 2e-16 ***
## white                   -0.019553   0.004465  -4.380 1.19e-05 ***
## factor(policePrecinct)2  0.008556   0.006757   1.266   0.2054
## factor(policePrecinct)3  0.003409   0.006483   0.526   0.5990
## factor(policePrecinct)4  0.084639   0.006232  13.582  < 2e-16 ***
## factor(policePrecinct)5 -0.012465   0.006371  -1.956   0.0504 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.254 on 19078 degrees of freedom
```

Case study: Parallel processing

Compute bootstrap clustered standard errors.

```r
# load packages
library(data.table)
# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)

# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)
# draw bootstrap samples, estimate model for each sample
for (i in 1:B) {

     # draw sample of precincts (cluster level)
     precincts_i <- sample(precincts, size = 5, replace = TRUE)
     # get observations
     bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
     bs_i <- rbindlist(bs_i)

     # estimate model and record coefficients
     boot_coefs[i,] <- coef(lm(model, bs_i))[1:2] # ignore FE-coefficients
}
```

Case study: Parallel processing

Finally, let's compute SE_{boot}.

```
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## [1] 0.004042725 0.004689611
```

Case study: Parallel processing

Parallel implementation…

```r
# install.packages("doSNOW", "parallel")
# load packages for parallel processing
library(doSNOW)
# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)

# get the number of cores available
ncores <- parallel::detectCores()
# set cores for parallel processing
ctemp <- makeCluster(ncores) #
registerDoSNOW(ctemp)


# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)

# bootstrapping in parallel
boot_coefs <-
     foreach(i = 1:B, .combine = rbind, .packages="data.table") %dopar% {

          # draw sample of precincts (cluster level)
```

Case study: Parallel processing

As a last step, we compute again SE_{boot}.

```r
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## (Intercept)       white
## 0.004416276 0.005102602
```

Case study: Memory allocation

```r
###########################################################
# Big Data Statistics: Flights data import and preparation
#
# U. Matter, January 2019
###########################################################

# SET UP -----------------

# fix variables
DATA_PATH <- "../data/flights.csv"

# DATA IMPORT ----------------
flights <- read.csv(DATA_PATH)

# DATA PREPARATION --------
flights <- flights[,-1:-3]
```

Case study: Memory allocation

Inspect the memory usage.

```
# SET UP -----------------

# fix variables
DATA_PATH <- "../data/flights.csv"
# load packages
library(pryr)
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
##
## Attaching package: 'pryr'
```

```
## The following object is masked from 'package:data.table':
##
##     address
```

```
# check how much memory is used by R (overall)
mem_used()
```

```
## 1.04 GB
```

Case study: Memory allocation

‘Collect the garbage’…

```
gc()
```

```
##              used  (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    1063841  56.9    1759040   94.0   1759040   94.0
## Vcells  126661755 966.4  213343342 1627.7 211037840 1610.1
```

Case study: Memory allocation

Alternative approach (via memory mapping).

```r
# load packages
library(data.table)

# DATA IMPORT ----------------
flights <- fread(DATA_PATH, verbose = TRUE)
```

```
##   OpenMP version (_OPENMP)       201511
##   omp_get_num_procs()            12
##   R_DATATABLE_NUM_PROCS_PERCENT  unset (default 50)
##   R_DATATABLE_NUM_THREADS        unset
##   R_DATATABLE_THROTTLE           unset (default 1024)
##   omp_get_thread_limit()         2147483647
##   omp_get_max_threads()          12
##   OMP_THREAD_LIMIT               unset
##   OMP_NUM_THREADS                unset
##   RestoreAfterFork               true
##   data.table is using 6 threads with throttle==1024. See ?setDTthreads.
## Input contains no \n. Taking this to be a filename to open
## [01] Check arguments
##   Using 6 threads (omp_get_max_threads()=12, nth=6)
##   NAstrings = [<<NA>>]
##   None of the NAstrings look like numbers.
##   show progress = 0
##   0/1 column will be read as integer
```

Case study: Memory allocation

Alternative approach (via memory mapping).

```
# SET UP -----------------

# fix variables
DATA_PATH <- "../data/flights.csv"
# load packages
library(pryr)
library(data.table)

# housekeeping
flights <- NULL
gc()
```

```
##             used  (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   1052878  56.3    1759040   94.0   1759040   94.0
## Vcells 123496007 942.2  213343342 1627.7 211037840 1610.1
```

```
# check the change in memory due to each step

# DATA IMPORT ----------------
mem_change(flights <- fread(DATA_PATH))
```

```
## 35.8 MB
```

References

Dhillon, Paramveer, Yichao Lu, Dean P. Foster, and Lyle Ungar. 2013. “New Subsampling Algorithms for Fast Least Squares Regression.” In Advances in Neural Information Processing Systems 26, 360–68.

Murrell, Paul. 2009. Introduction to Data Technologies. London, UK: CRC Press.