THE BIG R-BOOK:
FROM DATA SCIENCE TO BIG DATA AND LEARNING MACHINES

♡— PART: Bigger and Faster R —♡

by Philippe J.S. De Brouwer
for Wiley
version: 0.1

These slides are intended to be used in conjunction with the book: teachers will have read the book carefully before using the slides and students have access to the book and the code.

part: Bigger and Faster R

↓

chapter:

Parallel computing

The Library parallel Is Part Of Base R I

```
# Load the library:
library(parallel)

# The function detectCores finds the number of cores available:
numCores <- detectCores()
numCores
## [1] 12
```

Spreak Calculations over Multiple Cores I

```r
# Set the sample size:
N <- 100

# Set the starting points for the k-means algorithm:
starts <- rep(100, N)   # 50 times the value 100

# Load the dataset of the Titanic disaster:
library(titanic)

# Prepare data as numeric only:
t <- as.data.frame(titanic_train)
t <- t[complete.cases(t),]
t <- t[,c(2,3,5,6,7,8)]
t$Sex <- ifelse(t$Sex == 'male', 1, 0)

# Prepare the functions to be executed:
f <- function(nstart) kmeans(t, 4, nstart = nstart)

# Now, we are ready to go, we try two approaches:

# 1. With the standard function base::lapply
system.time(results <- lapply(starts, f))
##    user  system elapsed
##   1.912   0.000   1.912

# 2. With parallel::mclapply
system.time(results <- mclapply(starts, f, mc.cores = numCores))
##    user  system elapsed
##   3.298   0.139   0.346
```

part: Bigger and Faster R

↓

chapter: Parallel computing

↓

section:

Combine foreach and doParallel

foreach

```
# 1. The regular for loop:
for (n in 1:4) print(gamma(n))
## [1] 1
## [1] 1
## [1] 2
## [1] 6

# 2. The foreach loop:
library(foreach)
foreach (n = 1:4) %do% print(gamma(n))
## [1] 1
## [1] 1
## [1] 2
## [1] 6
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 6
```

doParallel COMBINED WITH foreach

```
# Once installed, the package must be loaded once per session:
library(doParallel)

# Find the number of available cores (as in previous section):
numCores <- parallel::detectCores()

# Register doParallel:
registerDoParallel(numCores)

# Now, we are ready to put it in action:
foreach (n = 1:4) %dopar% print(gamma(n))
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 6
```

Specify the Output Type

```
# Collapse to wide data.frame:
foreach (n = 1:4, .combine = cbind) %dopar% print(gamma(n))
##      result.1 result.2 result.3 result.4
## [1,]        1        1        2        6

# Collapse to long data.frame:
foreach (n = 1:4, .combine = rbind) %dopar% print(gamma(n))
##          [,1]
## result.1    1
## result.2    1
## result.3    2
## result.4    6

# Collapse to vector:
foreach (n = 1:4, .combine = c)     %dopar% print(gamma(n))
## [1] 1 1 2 6
```

part: Bigger and Faster R

↓

chapter: Parallel computing

↓

section:

Distribute Calculations over LAN with snow

Loading snow

We had loaded `parallel` in previous section, and the output shows how many of the functions overlap in both packages. This means two things. First, they should not be used together, they do the same but in different ways. Second, the similarity between both will shorten the total learning time.
The better approach is to unload `parallel` first:

```
detach("package:parallel", unload = TRUE)
library(snow)
```

This will create a cluster on the same machine.

```
cl <- makeCluster(c("localhost", "localhost"), type = "SOCK")
```

In this particular case, it is of course possible to uses the function's defaults and `makeCluster(2)` will have the same effect. However, remember that this functionality only makes sense when you use other machines than "localhost."
Now, that the cluster is defined, we can run operations over the cluster. We will use the same example as we used in previous section.

Connecting and Using snow II

```
library(titanic)  # provides the dataset: titanic_train
N <- 50
starts <- rep(100, N)   # 50 times the value 100

# Prepare data as numeric only:
t <- as.data.frame(titanic_train)
t <- t[complete.cases(t),]
t <- t[,c(2,3,5,6,7,8)]
t$Sex <- ifelse(t$Sex == 'male', 1, 0)

# Prepare the functions to be executed:
f <- function(nstart) kmeans(t, 4, nstart=nstart)

# 1. with the standard function
system.time(results <- lapply(starts, f))
##    user  system elapsed
##   0.969   0.000   0.969

# 2. with the cluster
# First, we must export the object t,  so that it can
# be used by the cluster:
clusterExport(cl, "t")
#clusterExport(cl, "starts") # Not needed since it is in the function f
system.time(
  result2 <- parLapply(cl, starts, f)
  )
##    user  system elapsed
##   0.009   0.000   0.515
```

Base R	**snow**
lapply	parLapply
sapply	parSapply
vapply	NA
apply (row-wise)	parRapply or parApply(,1)
apply (column-wise)	parCapply or parApply(,2)

TABLE 1: Base R functions and their alternative in snow.

Using and UDF in snow and Random Numbers I

```
f <- function (x, y) x + y + rnorm(1)
clusterCall(cl, function (x, y) {x + y + rnorm(1)}, 0, pi)
## [[1]]
## [1] 2.382391
##
## [[2]]
## [1] 1.956944

clusterCall(cl, f, 0, pi)
## [[1]]
## [1] 2.570099
##
## [[2]]
## [1] 4.529458

# Both forms are semantically similar to:
clusterCall(cl, eval, f(0,pi))
## [[1]]
## [1] 4.063712
##
## [[2]]
## [1] 4.063712

# However, note that the random numbers are the same on both clusters.
```

The cluster version of the function `evalq()` is `clusterCallEvalQ()` and it is called as follows.

Using and UDF in snow and Random Numbers II

```
# Note that ...
clusterEvalQ(cl, rnorm(1))
## [[1]]
## [1] 0.6814072
##
## [[2]]
## [1] -0.5958536

# ... is almost the same as
clusterCall(cl, evalq, rnorm(1))
## [[1]]
## [1] -0.0127988
##
## [[2]]
## [1] -0.0127988

# ... but that the random numbers on both slaves nodes are the same
```

part: Bigger and Faster R

↓

chapter: Parallel computing

↓

section:

Using the GPU

```
sudo lshw -numeric -C display
```

The output can look as follows:

```
*-display
       description: VGA compatible controller
       product: Crystal Well Integrated Graphics Controller [8086:D26]
       vendor: Intel Corporation [8086]
       physical id: 2
       bus info: pci@0000:00:02.0
       version: 08
       width: 64 bits
       clock: 33MHz
       capabilities: msi pm vga_controller bus_master cap_list rom
       configuration: driver=i915 latency=0
       resources: irq:30 memory:f7800000-f7bfffff memory:e0000000-efffffff ioport
     :f000(size=64) memory:c0000-dffff
```

If you have a dedicated GPU, the output will provide information about the GPU:

```
  *-display
description: VGA compatible controller
product: NVIDIA Corporation [10DE:2191]
vendor: NVIDIA Corporation [10DE]
physical id: 0
bus info: pci@0000:01:00.0
version: a1
width: 64 bits
clock: 33MHz
capabilities: pm msi pciexpress vga_controller bus_master cap_list rom
configuration: driver=nvidia latency=0
resources: irq:154 memory:b3000000-b3ffffff memory:a0000000-afffffff memory:
     b0000000-b1ffffff ioport:4000(size=128) memory:c0000-dffff
```

Using the GPU From R via gpuR: Define Large Matrices

```
# Install.packages("gpuR")  # do only once
library(gpuR)               # load gpuR
## Number of platforms: 1
## - platform: NVIDIA Corporation: OpenCL 1.2 CUDA 10.1.0
##   - context device index: 0
##     - GeForce GTX 1660 Ti
## checked all devices
## completed initialization

## gpuR 2.0.3
## Attaching package: 'gpuR'
## The following objects are masked from 'package:base':
## colnames, pmax, pmin, svd

detectGPUs()                # check if it all works
## [1] 1

# Prepare an example:
N <- 516
A <- matrix(rnorm(N^2), nrow = N, ncol = N)
gpuA <- gpuMatrix(A)        # prepare the matrix to be used on GPU


# In base R we could do this:
B <- A %*% A

# gpuR works as one would expect:
gpuB <- gpuA %*% gpuA

# note its structure:
gpuB
## An object of class "fgpuMatrix"
## Slot "address":
```

First Simple Results are Very Encocuraging

```
# base R:
system.time(B <- A %*% A)
##    user  system elapsed
##   0.052   0.000   0.051

# gpuR works exactly as one would expect:
system.time(gpuB <- gpuA %*% gpuA)
##    user  system elapsed
##   0.022   0.000   0.023
```

Plotting the Time Gain In Function of Matrix Size I

```
set.seed(1890)
NN <- seq(from = 500, to = 4500,by = 1000)
t  <- data.frame(N = numeric(), CPU = numeric(), GPU = numeric())
i  <- 1
for(k in NN) {
  A <- matrix(rnorm(k^2), nrow = k, ncol = k)
  gpuA <- gpuMatrix(A)
  t[i,1] <- k
  t[i,2] <- system.time(B <- A %*% A)[3]
  t[i,3] <- system.time(gpuB <- gpuA %*% gpuA)[3]
  i <- i + 1
}
# Print the results
t
##      N    CPU   GPU
## 1  500  0.047 0.006
## 2 1500  1.597 0.057
## 3 2500  8.634 0.285
## 4 3500 22.995 0.686
## 5 4500 48.545 1.094

# Tidy up the data-frame:
library(tidyr)
tms <- gather(t, 'PU', 'time', 2:3)
```

Plotting the Time Gain In Function of Matrix Size II

```
# Plot the results:
library(ggplot2)
p <- ggplot(tms, aes(x = N, y = time, colour = PU)) +
     geom_point(size=5) +
     geom_line() +
     xlab('Matrix size (number of rows and columns)') +
     ylab('Time in seconds') +
     theme(axis.text = element_text(size=12),
           axis.title = element_text(size=14)
           )
print(p)
```

Plotting the Time Gain In Function of Matrix Size III

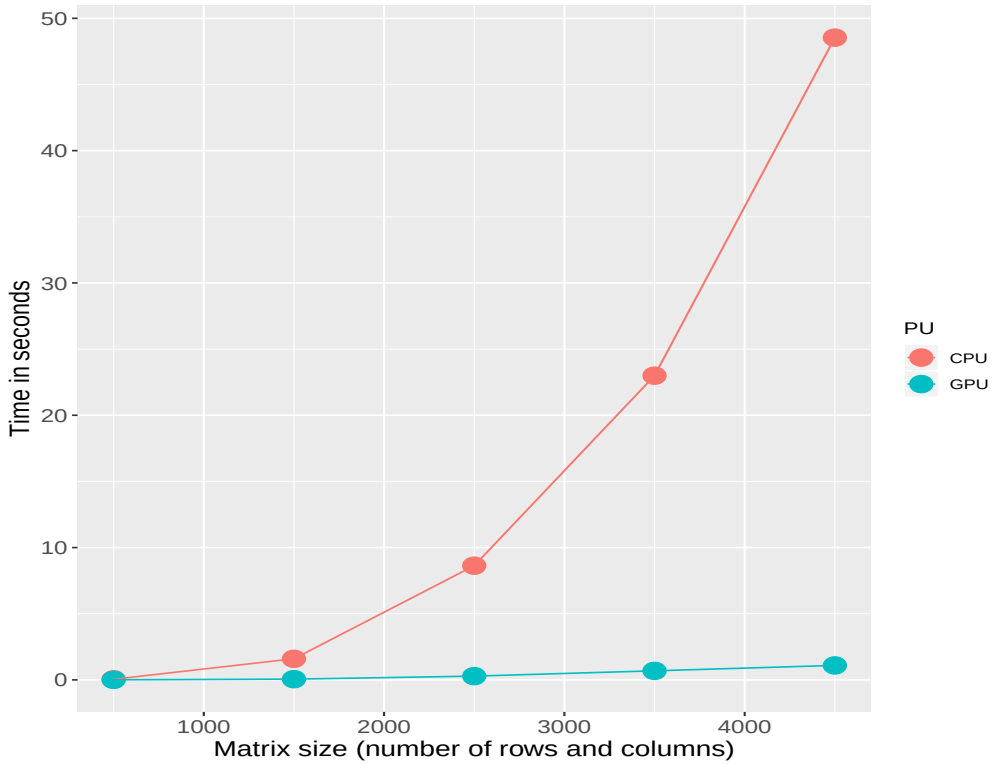

FIGURE 1: The runtimes for matrix multiplication compared on the CPU versus the GPU.

Other Functions Available in gpuR I

The usability of a GPU programming library depends on the calculations that one can spread over the cores of the graphics card. The library `gpuR` also provides methods for basic arithmetic and calculus such as: `%*%`, `+`, `-`, `*`, `/`, `t`, `crossprod`, `tcrossprod`, `colMeans`, `colSums`, `rowMeans`, `rowSums`, `sin`, `asin`, `sinh`, `cos`, `acos`, `cosh`, `tan`, `atan`, `tanh`, `exp`, `log`, `exp`, `abs`, `max`, `min`, `cov`, `eigen`. There are many other operations available such as Euclidian distance, etc. With all those functions `gpuR` is one of the most complete, but it is also one of the most universal GPU programming options available for the R programmer.

When a Matrix is Re-Used and the GPU has Memory: Store It There I

```
require(gpuR)
require(tidyr)
require(ggplot2)
set.seed(1890)
NN <- seq(from = 500, to = 5500,by = 1000)
t  <- data.frame(N = numeric(), `CPU/CPU` = numeric(),
                 `CPU/GPU` = numeric(), `GPU/GPU` = numeric())
i <- 1


# Run the experiment
for(k in NN) {
A <- matrix(rnorm(k^2), nrow = k, ncol = k)
                    # storage in CPU-RAM, calculations in CPU
gpuA <- gpuMatrix(A) # storage in CPU-RAM, calculations in GPU
vclA <- vclMatrix(A) # storage in GPU-RAM, calculations in GPU
t[i,1] <- k
t[i,2] <- system.time(B <- A %*% A)[3]
t[i,3] <- system.time(gpuB <- gpuA %*% gpuA)[3]
t[i,4] <- system.time(vclB <- vclA %*% vclA)[3]
i <- i + 1
}
```

When a Matrix is Re-Used and the GPU has Memory: Store It There II

```
# Print the results
t
##      N CPU.CPU CPU.GPU GPU.GPU
## 1  500   0.047   0.005   0.004
## 2 1500   1.635   0.054   0.018
## 3 2500   8.088   0.244   0.003
## 4 3500  22.172   0.577   0.003
## 5 4500  47.235   1.111   0.003
## 6 5500  86.114   1.843   0.003
```

When a Matrix is Re-Used and the GPU has Memory: Store It There III

```
# Tidy up the data-frame:
tms <- gather(t, 'RAM/PU', 'time', 2:4)

# Plot the results:
scaleFUN <- function(x) sprintf("%.2f", x)

p <- ggplot(tms, aes(x = N, y = time, colour = `RAM/PU`)) +
  geom_point(size=5) +
  geom_line() +
  scale_y_continuous(trans = 'log2', labels = scaleFUN) +  # NEW!!
  xlab('Matrix size (number of rows and columns)') +
  ylab('Time in seconds') +
  theme(axis.text = element_text(size=12),
  axis.title = element_text(size=14)
)
print(p)
```

When a Matrix is Re-Used and the GPU has Memory: Store It There IV

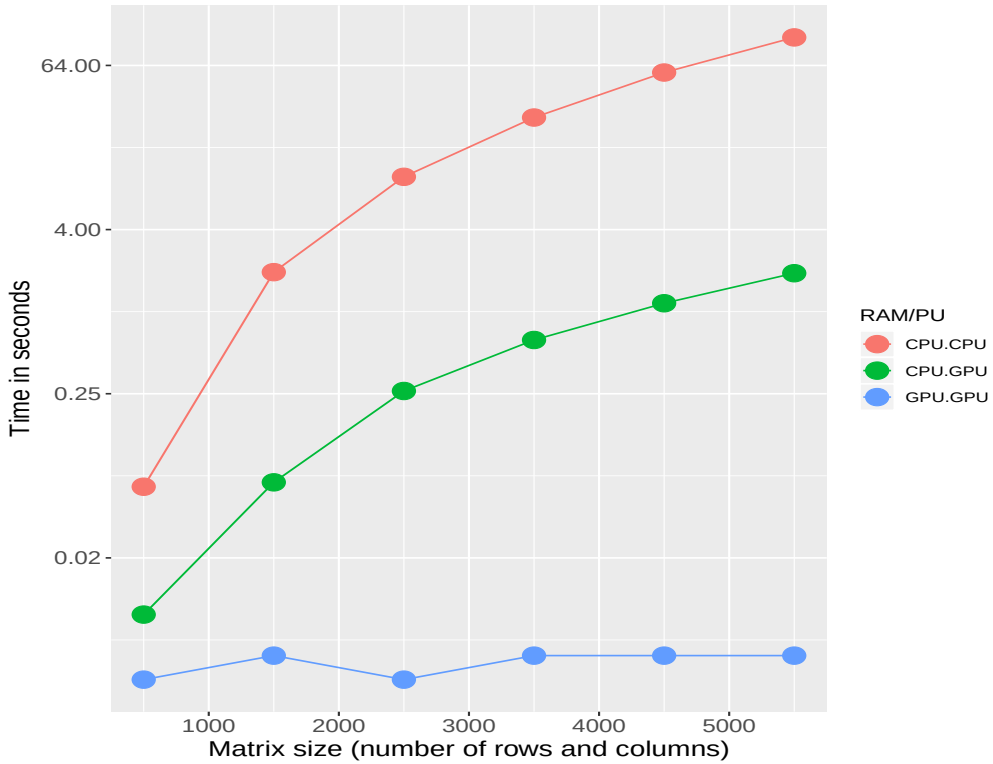

Figure 2: The runtimes for matrix multiplication for storage/calculation pairs: CPU/CPU, CPU/GPU, GPU/GPU. Note that the scale of the *y*-axisis logarithmic.

```
# unload gpuR:
detach('package:gpuR', unload = TRUE)
```

part: Bigger and Faster R

↓

chapter:

R and Big Data

Big Data Defined

- velocity: the speed at which the data comes in,
- variety: the number of columns and formats,
- veracity: the reliability or data quality, and
- volume: the amount of data.

Strategies in R to Solve Big Data Problems I

1. Reduce the size of the data in the source system.
2. Take a sample of the data that can be handled, make the model and then calculate the model performance on the whole table.
3. Use the most powerful computer at your disposal.
4. Avoid loading the whole table at once in RAM, and leave most on the HDD via `ff`, `Biglm`, `RODBC`, `snow`, etc.
5. Use parallelism in data-storage and calculations.

part: Bigger and Faster R

↓

chapter: R and Big Data

↓

section:

Use a Powerful Server

It is also possible to use R as a service that runs on a server. R can be installed on the server and you can login remotely or use `ssh` to open a terminal that is visible on your screen but only reflects the CLI of the server. With R installed there, you can access the data that sits there and potentially benefit of the large array of disks that is probably expressed in giga-bytes and maybe even in peta-bytes.

Hint – RStudio

If you do not like the command line too much, then you might want to check "RStudio Server." It provides a browser based interface to a version of R running on a remote Linux server. So, it will look as if you use the RStudio IDE locally, but actually you are using the power and storage capacity of the server.

Delegate Data Wrangling to the SQL Server

Most of the data-wrangling can be done directly on the server or via R. If we choose to do this via R, then there are largely two approaches.

1. Connect to the server.
2. Using the library `dplyr` to generate the SQL code for you and execute the query only when needed.

part: Bigger and Faster R

↓

chapter: R and Big Data

↓

section:

Using more memory than we have RAM

Use Datasets Larger than the Available RAM

To the library `ff` provides data-structures that are stored on the disk rather than in RAM. When the data is needed, it will be loaded chunk after chunk.
There are also a few other packages that do the same, while targeting specific use. For example:

- `bigmemory` for large matrices,
- `biglm` for building generalised linear models.

part: Bigger and Faster R

↓

chapter:

Parallelism for Big Data

In 2004 Google published the seminal paper in which they described the process "MapReduce" that allows for a parallel processing model, to process huge amounts of data in parallel nodes. MapReduce will split queries and distributes them over the parallel nodes, where they are processed simultaneously (in parallel). This phase is called the "Map step." Once a node has obtained its results, it will pass them on to the layer higher. This is the "Reduce step," where the results are stitched together for the user.
This approach has many advantages:

- it is able to handle massive amounts of data in incredible speeds,
- because of the built-in redundancy it is very reliant and resilient,
- therefore, it is also possible to use cheaper hardware in the nodes (if one node is busy or not online, then the task will be dispatched to the alternate CPU that also has the same part of the data), and it becomes also a cost efficient solution.

part: Bigger and Faster R

↓

chapter: Parallelism for Big Data

↓

section:

Apache Hadoop

The core of Hadoop is known as “Hadoop Common” and consists of:

- `HDFS` is the Hadoop Distributed File System is the lowest level of the ecosystem: it manages the physical storage of the distributed files in a redundant way.
- `Hadoop YARN` takes care of the cluster resource management.
- `Hadoop MapReduce` is the mapping and reducing engine.

Also in the Hadoop Ecosystem

The family of software that populate the Hadoop ecosystem is much larger than Hadoop Common. Here are some of the solutions that usually are found together.

- `Zookeeper:` is a high availability coordination and configuration service for distributed systems.
- `Apache Spark:` is an in-memory data-flow engine: it is a cluster computing framework for massive distributed data and provides an interface that makes abstraction of the data parallelism with built-in fault tolerance.
- `Apache Storm:` is the computation framework for the distributed stream processing.
- `Oozie:` is the workflow scheduling system that manages different Hadoop jobs for the client.
- *Query and data intelligence interfaces* that are designed to allow for massively parallel processing.
  - `Apache Pig:` allows to create programs that run on Hadoop via the high-level language "Pig Latin."
  - `HBase:` the non relational distributed database.
  - `Apache Phoenix:` the relational database engine (using HBase underneath).
  - `Apache Impala:` the SQL query engine.
  - `Mahout` and `Spark MLib:` machine learning libraries.
  - `Drill:` for interactive analysing data.
- `Sqoop:` allows to import data from relational databases
- `Apache Flume` takes care of the massive amounts of log-data.

part: Bigger and Faster R

↓

chapter: Parallelism for Big Data

↓

section:

Apache Spark

Within the Hadoop project and the wider Hadoop ecosystem, Spark takes a special place with the resilient distributed dataset concept (RDD). It is a read-only equivalent of the data-frame in R, but it is designed to keep the dataset distributed over a cluster of machines.

Apache Spark is an open-source distributed cluster-computing framework for general use (it is not specialised per se). Spark also provides a logical and easy-to-use interface for using clusters with implicit data parallelism and fault tolerance. This means that the programmer does not have to manage the parallelism or fault tolerance: this is done in the background.

Notice: Spark Assumed in the Remainder of This Chapter

For the rest of this chapter we will assume that you and your computer have access to a cluster that runs Spark.

Digression – Spark installer

We thank RStudio and Microsoft to have created and made available a useful cross-platform installer for Apache Spark. This installer is designed to use system resources efficiently under a common API. It will support as well R as Python.

```
# install from github
devtools::install_github(repo = "rstudio/spark-install",
                         subdir = "R")
library(sparkinstall)

# lists the versions available to install
spark_available_versions()

# installs an specific version
spark_install(version = "2.4.3")

# uninstalls an specific version
spark_uninstall(version = "2.4.3", hadoop_version = "2.6")
```

This can be done with the command:

```
start-master.sh
```

The system will reply with the location of the log-file and return to the command prompt. It might seem that not too much happened, but Spark is really active on the system now and ready to reply to instructions. We can use the CLI to interrogate the system status or use a web-browser and point it to `http://localhost:8080`. The browser-interface will look similar to the one in Figure 3 on slide 46.

Starting the Spark Cluster II

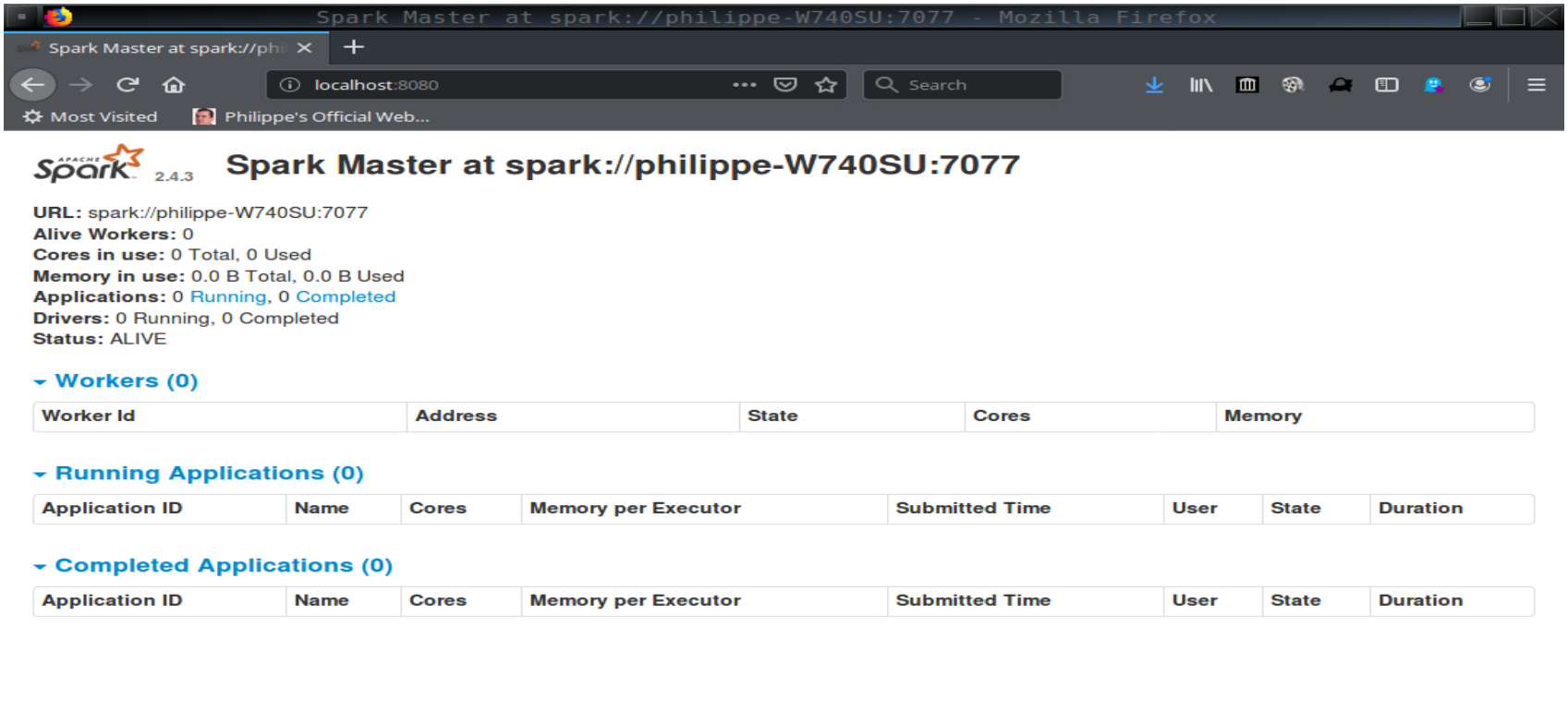

Figure 3: The status of Spark can be controlled via a regular web-browser, that is directed to `http://localhost:8080`.

There is – of course – no need to use a web-browser nor even have a windows manager active: the status of Spark can also be checked via the CLI. This can be done via the command `ss`.

Starting the Spark Cluster III

```
$ ss -tunelp | grep 8080
tcp    LISTEN     0      1         :::8080                 :::*
     users:(("java",pid=26295,fd=351)) uid:1000 ino:5982022 sk:1d v6only:0 <->
```

Now, we are sure that the Spark-master is up and running and can be used on our system. So, we can start a Spark-slave via the shell command `start-slave.sh` or via the Spark-shell. This shell can be invoked via the command `spark-shell` and will look as follows.

```
$ spark-shell
19/07/14 21:16:43 WARN Utils: Your hostname, philippe-W740SU resolves to a
     loopback address: 127.0.1.1; using 192.168.100.120 instead (on interface
     enp0s25)
19/07/14 21:16:43 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
     address
19/07/14 21:16:44 WARN NativeCodeLoader: Unable to load native-hadoop library for
      your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel
     (newLevel).
Spark context Web UI available at http://192.168.100.120:4040
Spark context available as 'sc' (master = local[*], app id = local-1563131811133)
     .
Spark session available as 'spark'.
Welcome to
      ____              __
```

```
      / __/__  ___ _____/ /__
     _\ \/ _ \/ _ `/ __/  '_/
    /___/ .__/\_,_/_/ /_/\_\   version 2.4.3
       /_/

Using Scala version 2.11.12 (IBM J9 VM, Java 1.8.0_211)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Read this welcome message, because it is packed with useful information. For example, it refers you to a web user interface (in our case at `http://192.168.100.120:4040`). Note that when you work on the computer that has the Spark-master installed and running, that this is always equivalent to `http://localhost:4040`. The screen of this page is shown in Figure 4 on slide 49

Starting the Spark Cluster V

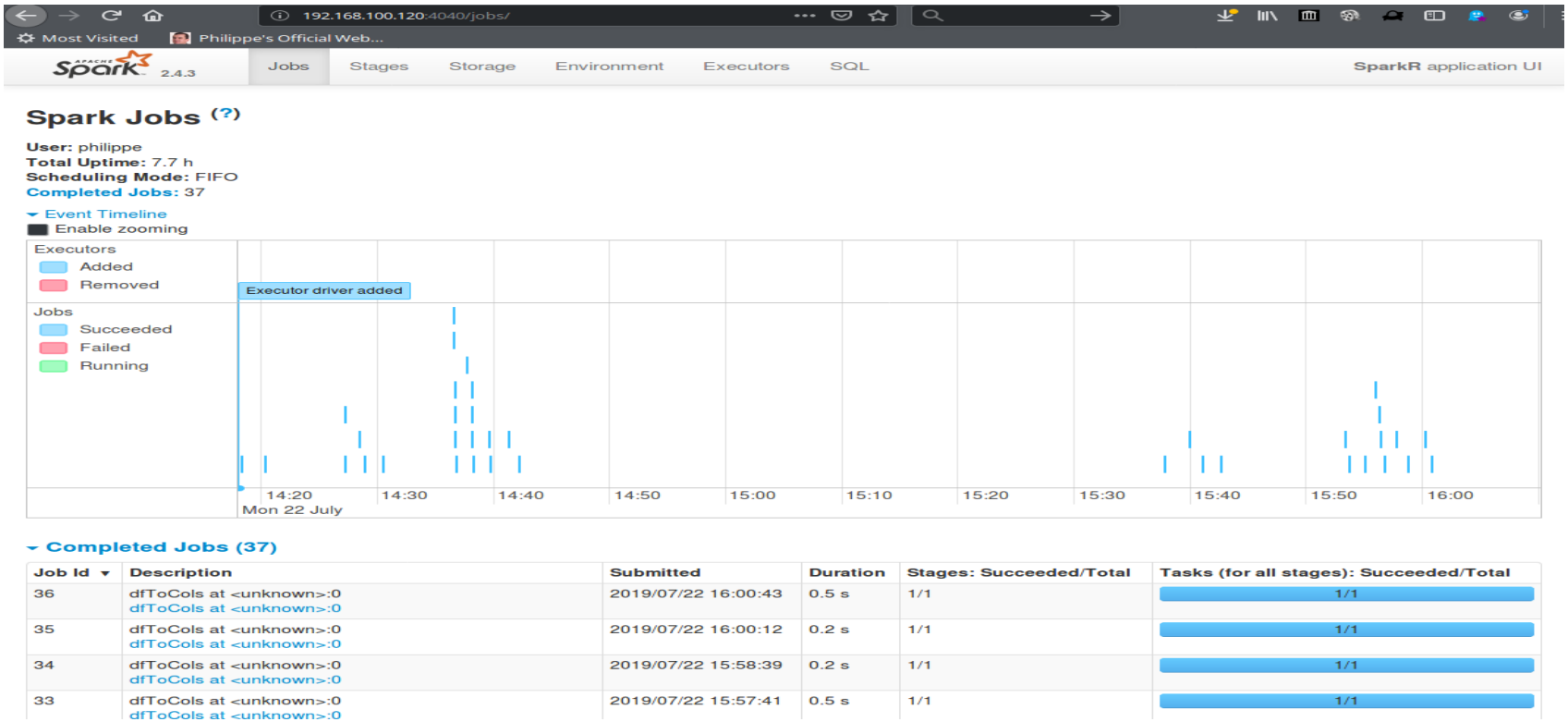

FIGURE 4: Each user can check information about his own Spark connection via a web-browser directed to `http://localhost:4040`. Here we show the "jobs" page after running some short jobs in the Spark environment.

Starting SparkR and Addressing the SparkDataFrame

```
library(tidyverse)
library(dplyr)
library(SparkR)
# Note that loading SparkR will generate many warning messages,
# because it overrrides many functions such as summary, first,
# last, corr, ceil, rbind, expr, cov, sd and many more.

sc <- sparkR.session(master = "local", appName = 'first test',
                     sparkConfig = list(spark.driver.memory = '2g'))
## Launching java with spark-submit command /opt/spark/bin/spark-submit   --driver-memory "2g" sparkr-shell /tmp/RtmpxRzeZ

# Show the session:
sc
## Java ref type org.apache.spark.sql.SparkSession id 1
```

```
# Create a SparkDataFrame:
DF <- as.DataFrame(mtcars)

# The DataFrame is for big data, so the attempt to print all data,
# might surprise us a little:
DF
## SparkDataFrame[mpg:double, cyl:double, disp:double, hp:double, drat:double, wt:double, qsec:double, vs:double, am:doubl

# R assumes that the data-frame is big data and does not even
# start printing all data.

# head() will collapse the first lines to a data.frame:
head(DF)
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

An Example for the Remainder of this Section

To develop an example, we will use the well known database `titanic`.
In the following code, we create the `DataFrame` and show some of its properties.

```
library(titanic)
library(tidyverse)

# This provides a.o. two datasets titanic_train and titanic_test.
# We will work further with the training-dataset:
T <- as.DataFrame(titanic_train)
```

Some Properties of our New SparkDataFrame

```
# The SparkDataFrame inherits from data.frame, so most functions
# work as expected on a DataFrame:
colnames(T)
##  [1] "PassengerId" "Survived"    "Pclass"      "Name"
##  [5] "Sex"         "Age"         "SibSp"       "Parch"
##  [9] "Ticket"      "Fare"        "Cabin"       "Embarked"

str(T)
## 'SparkDataFrame': 12 variables:
##  $ PassengerId: int 1 2 3 4 5 6
##  $ Survived   : int 0 1 1 1 0 0
##  $ Pclass     : int 3 1 3 1 3 3
##  $ Name       : chr "Braund, Mr. Owen Harris" "Cumings, Mrs. John Bradley (Florence Briggs Thayer)" "Heikkinen, Miss. La
##  $ Sex        : chr "male" "female" "female" "female" "male" "male"
##  $ Age        : num 22 38 26 35 35 NA
##  $ SibSp      : int 1 1 0 1 0 0
##  $ Parch      : int 0 0 0 0 0 0
##  $ Ticket     : chr "A/5 21171" "PC 17599" "STON/O2. 3101282" "113803" "373450" "330877"
##  $ Fare       : num 7.25 71.2833 7.925 53.1 8.05 8.4583
##  $ Cabin      : chr "" "C85" "" "C123" "" ""
##  $ Embarked   : chr "S" "C" "S" "S" "S" "Q"

summary(T)
## SparkDataFrame[summary:string, PassengerId:string, Survived:string, Pclass:string, Name:string, Sex:string, Age:string, SibSp:string, Parch:string, Ticket:strin

class(T)
## [1] "SparkDataFrame"
## attr(,"package")
## [1] "SparkR"

# The scheme is a declaration of the structure:
printSchema(T)
## root
##  |-- PassengerId: integer (nullable = true)
##  |-- Survived: integer (nullable = true)
##  |-- Pclass: integer (nullable = true)
##  |-- Name: string (nullable = true)
##  |-- Sex: string (nullable = true)
##  |-- Age: double (nullable = true)
##  |-- SibSp: integer (nullable = true)
##  |-- Parch: integer (nullable = true)
##  |-- Ticket: string (nullable = true)
##  |-- Fare: double (nullable = true)
##  |-- Cabin: string (nullable = true)
##  |-- Embarked: string (nullable = true)

# Truncated information collapses to data.frame:
T %>% head %>% class
## [1] "data.frame"
```

Most dplyr Functions Are Available in SparkR I

For example, selecting a row can be done in a very similar way as in base-R. The following lines of code do all the same: select one column of the Titanic data and then show the first ones.

```
X <- T %>% SparkR::select(T$Age)          %>% head
Y <- T %>% SparkR::select(column('Age')) %>% head
Z <- T %>% SparkR::select(expr('Age'))   %>% head
cbind(X, Y, Z)
##   Age Age Age
## 1  22  22  22
## 2  38  38  38
## 3  26  26  26
## 4  35  35  35
## 5  35  35  35
## 6  NA  NA  NA
```

The package dplyr offers SQL-like functions to manipulate and select data, and this functionality works also on a DataFrame using SparkR. In many cases, this will allow us to reduce the big data problem to a small data problem that can be used for analysis. For example, we can select all young males that survived the Titanic disaster as follows:

Most dplyr Functions Are Available in SparkR II

```
T %>%
  SparkR::filter("Age < 20 AND Sex == 'male' AND Survived == 1") %>%
  SparkR::select(expr('PassengerId'), expr('Pclass'), expr('Age'),
                 expr('Survived'), expr('Embarked'))             %>%
  head
##   PassengerId Pclass   Age Survived Embarked
## 1          79      2  0.83        1        S
## 2         126      3 12.00        1        C
## 3         166      3  9.00        1        S
## 4         184      2  1.00        1        S
## 5         194      2  3.00        1        S
## 6         205      3 18.00        1        S

# The following is another approach. The end-result is the same, however,
# we bring the data first to the R's working memory and then use dplyr.
# Note the subtle differences in syntax.
SparkR::collect(T)                                                %>%
      dplyr::filter(Age < 20 & Sex == 'male' & Survived == 1)     %>%
      dplyr::select(PassengerId, Pclass, Age, Survived,
                    Embarked)                                     %>%
      head
##   PassengerId Pclass   Age Survived Embarked
## 1          79      2  0.83        1        S
## 2         126      3 12.00        1        C
## 3         166      3  9.00        1        S
## 4         184      2  1.00        1        S
## 5         194      2  3.00        1        S
## 6         205      3 18.00        1        S
```

SparkR has its functions modelled to dplyr and most functions will work as expected. For example, grouping, summarizing, changing and arranging data works as in dplyr.

Most dplyr Functions Are Available in SparkR III

```
# Extract the survival percentage per class for each gender:
TMP <- T                                                    %>%
       SparkR::group_by(expr('Pclass'), expr('Sex'))        %>%
             summarize(countS = sum(expr('Survived')), count = n(expr('PassengerId')))
N   <- nrow(T)

TMP                                                                 %>%
    mutate(PctAll   = expr('count') / N  * 100)                     %>%
    mutate(PctGroup = expr('countS') / expr('count')  * 100)        %>%
    arrange('Pclass', 'Sex')                                        %>%
    SparkR::collect()
##   Pclass    Sex countS count    PctAll PctGroup
## 1      1 female     91    94 10.549944 96.80851
## 2      1   male     45   122 13.692480 36.88525
## 3      2 female     70    76  8.529742 92.10526
## 4      2   male     17   108 12.121212 15.74074
## 5      3 female     72   144 16.161616 50.00000
## 6      3   male     47   347 38.945006 13.54467
```

Run A User Defined Function on a SparkDataFrame with `dapply`

Running a user defined function (UDF) on a SparkDataFrame (a resilient distributed dataset) is remarkably simple with the function `dapply()`. The UDF is passed as a parameter to `dapply` and it should have only use one parameter: an R data.frame. The output must also be an R-data.frame and nothing else.

```
# The data:
T <- as.DataFrame(titanic_train)

# The schema can be a structType:
schema <- SparkR::structType(SparkR::structField("Age", "double"),
                             SparkR::structField("ageGroup", "string"))

# Or (since Spark 2.3) it can also be a DDL-formatted string
schema <- "age DOUBLE, ageGroup STRING"

# The function to be applied:
f <- function(x) {
  data.frame(x$Age, if_else(x$Age < 30, "youth", "mature"))
  }

# Run the function f on the Spark cluster:
T2 <- SparkR::dapply(T, f, schema)

# Inspect the results:
head(SparkR::collect(T2))

##     age           ageGroup
## 1    22              youth
## 2    38             mature
## 3    26              youth
## 4    35             mature
## 5    35             mature
```

dapplyCollect Collects the Results to an R-Data-Frame

The "collect" version, `dapplyCollect()`, will apply the provided user defined function to each partition of the SparkDataFrame and collect the results back into a familiar R data.frame. Note also that the schema does not have to be provided to the function `dapplyCollect()`, because it returns a `data.frame` and not a `DataFrame`.

```
# The data:
T <- as.DataFrame(titanic_train)

# The function to be applied:
f <- function(x) {
  y <- data.frame(x$Age, ifelse(x$Age < 30, "youth", "mature"))

  # We specify now column names in the data.frame to be returned:
    colnames(y) <- c("age", "ageGroup")

  # and we return the data.frame (base R type):
  y
  }

# Run the function f on the Spark cluster:
T2_DF <- dapplyCollect(T, f)

# Inspect the results (T2_DF is now a data.frame, no collect needed):
head(T2_DF)
##   age ageGroup
## 1  22    youth
## 2  38   mature
## 3  26    youth
## 4  35   mature
## 5  35   mature
## 6  NA     <NA>
```

The Group Apply Variant: gapply

```
# define the function to be used:
f <- function (key, x) {
  data.frame(key, min(x$Age,  na.rm = TRUE),
                  mean(x$Age, na.rm = TRUE),
                  max(x$Age,  na.rm = TRUE))
  }

# The schema also can be specified via a DDL-formatted string
schema <- "class INT, min_age DOUBLE, avg_age DOUBLE, max_age DOUBLE"

maxAge <- gapply(T, "Pclass", f, schema)

head(collect(arrange(maxAge, "class", decreasing = TRUE)))
##   class min_age  avg_age max_age
## 1     3    0.42 25.14062      74
## 2     2    0.67 29.87763      70
## 3     1    0.92 38.23344      80
```

gapplyCollect

The function `gapply` has a variant that "collects" the result to an R data.frame. The main difference in usage is that the schema does not need to be supplied.

```
# define the function to be used:
f <- function (key, x) {
  y <- data.frame(key, min(x$Age,  na.rm = TRUE),
                       mean(x$Age, na.rm = TRUE),
                       max(x$Age,  na.rm = TRUE))
  colnames(y) <- c("class", "min_age", "avg_age", "max_age")
  y
  }

maxAge <- gapplyCollect(T, "Pclass", f)

head(maxAge[order(maxAge$class, decreasing = TRUE), ])
##   class min_age  avg_age max_age
## 2     3    0.42 25.14062      74
## 3     2    0.67 29.87763      70
## 1     1    0.92 38.23344      80
```

spark.lapply is Equivalent to lapply

```
# First a trivial example to show how spark.lapply works:
surfaces <- spark.lapply(1:3, function(x){pi * x^2})
print(surfaces)
## [[1]]
## [1] 3.141593
##
## [[2]]
## [1] 12.56637
##
## [[3]]
## [1] 28.27433
```

The power of `spark.lapply` resides of course not in calculating a list of squares, but rather in executing more complex model fitting over large distributed datasets. Below we show how it can fit a model.

```
mFamilies   <- c("binomial", "gaussian")
trainModels <- function(fam) {
  m <- SparkR::glm(Survived ~ Age + Pclass,
                   data = T,
                   family = fam)
  summary(m)
  }
mSummaries <- spark.lapply(mFamilies, trainModels)
```

It is essential to be able to move data.frames to the distributed filesystem as a resilient distributed dataset (or SparkDataFrame) and it is also necessary to be able to collect the data back to R. The following code shows conversions in both directions.

```
# df is a data.frame (R-data.frame)
# DF is a DataFrame (distributed Spark data-frame)

# We already saw how to get data from Spark to R:
df <- collect(DF)

# From R to Spark:
DF <- createDataFrame(df)
```

Reading CSV files

Data can reside in SQL or NoSQL databases, but sometimes it is necessary to input plain textfiles such as CSV-files. This can be done as follows.

```
loadDF(fileName,
       source = "csv",
       header = "true",
       sep = ",")
```

Changing Columns and Adding New Ones

Apart from the functions `mutate()` and `select`, there are much more functions from `dplyr` re-engineered to work on a SparkDataFrame. For example, we can change and add columns via the function `withColumn`.

```
T %>% withColumn("AgeGroup", column("Age") / lit(10))                    %>%
      SparkR::select(expr('PassengerId'), expr('Age'), expr('AgeGroup')) %>%
      head
##   PassengerId Age AgeGroup
## 1           1  22      2.2
## 2           2  38      3.8
## 3           3  26      2.6
## 4           4  35      3.5
## 5           5  35      3.5
## 6           6  NA       NA
```

Note that in the code fragment above, the function `lit()` returns the literal value of 10.

In addition to the `group_by()` function used before, SparkR also provides the aggregation function `agg()`
as well as the OLAP cube operator: `cube()`.

```
T %>% cube("Pclass", "Sex")  %>%
      agg(avg(T$Age))        %>%
      collect()
##    Pclass    Sex avg(Age)
## 1       3   <NA> 25.14062
## 2       1   male 41.28139
## 3       2   male 30.74071
## 4       1   <NA> 38.23344
## 5       2 female 28.72297
## 6      NA   <NA> 29.69912
## 7       1 female 34.61176
## 8       2   <NA> 29.87763
## 9       3   male 26.50759
## 10     NA female 27.91571
## 11     NA   male 30.72664
## 12      3 female 21.75000
```

1. **Classification**
   - `spark.logit`: Logistic Regression
   - `spark.mlp`: Multilayer Perceptron (MLP)
   - `spark.naiveBayes`: Naive Bayes
   - `spark.svmLinear`: Linear Support Vector Machine
2. **Regression**
   - `spark.glm` or `glm`: Generalized Linear Model (GLM)
   - `spark.survreg`: Accelerated Failure Time (AFT) Survival Model
   - `spark.isoreg`: Isotonic Regression
3. **Tree**
   - `spark.gbt`: Gradient Boosted Trees for Regression and Classification
   - `spark.randomForest`: Random Forest for Regression and Classification
4. **Clustering**
   - `spark.kmeans`: K-Means
   - `spark.bisectingKmeans`: Bisecting *k*-means
   - `spark.gaussianMixture`: Gaussian Mixture Model (GMM)
   - `spark.lda`: Latent Dirichlet Allocation (LDA)
5. **Collaborative Filtering**
   - `spark.als`: Alternating Least Squares (ALS)
6. **Frequent Pattern Mining**
   - `spark.fpGrowth` : FP-growth
7. **Statistics**
   - `spark.kstest`: Kolmogorov-Smirnov Test

SparkR makes it possible to store results of models and retrieve them via the functions `write.ml()` and `read.ml()`.

```
# Prepare training and testing data:
T_split <- randomSplit(T, c(8,2), 2)
T_train <- T_split[[1]]
T_test  <- T_split[[2]]

# Fit the model:
M1 <- spark.glm(T_train, Survived ~ Pclass + Sex, family = "binomial")

# Save the model:
path1 <- tempfile(pattern = 'ml', fileext = '.tmp')
write.ml(M1, path1)
```

```
# Retrieve the model
M2 <- read.ml(path1)

# Do something with M2:
summary(M2)
##
## Saved-loaded model does not support output 'Deviance Residuals'.
##
## Coefficients:
##              Estimate  Std. Error   t value    Pr(>|t|)
## (Intercept)   3.30997     0.33860    9.7755  0.0000e+00
## Pclass       -0.98022     0.11989   -8.1758  4.4409e-16
## Sex_male     -2.62236     0.20723  -12.6542  0.0000e+00
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 932.42  on 704  degrees of freedom
## Residual deviance: 654.45  on 702  degrees of freedom
```

```
# Add predictions to the model for the test data:
PRED1 <- predict(M2, T_test)

# Show the results:
x <- head(collect(PRED1))
head(cbind(x$Survived, x$prediction))
##      [,1]      [,2]
## [1,]    0 0.4273654
## [2,]    1 0.9113117
## [3,]    0 0.2187741
## [4,]    0 0.4273654
## [5,]    0 0.4273654
## [6,]    0 0.4273654
```

```
# Close the connection:
unlink(path1)
```

A Worthy Alternative: SparklyR

Make sure to detach SparkR first and install the package sparklyr.

```
# First, load the tidyverse packages:
library(tidyverse)
library(dplyr)

# Load the package sparklyr:
library(sparklyr)

# Load the data:
library(titanic)

# Our spark-master is already running, so no need for:
# system('start-master.sh')

# Connect to the local Spark master
sc <- spark_connect(master = "local")
```

This connection object, `sc`, allows us to connect tot Spark as follows:

```
Titanic_tbl <- copy_to(sc, titanic_train)
Titanic_tbl

## # Source: spark<titanic_train> [?? x 12]
##    PassengerId Survived Pclass Name    Sex     Age SibSp Parch ...
##          <int>    <int>  <int> <chr>   <chr> <dbl> <int> <int> ...
##  1           1        0      3 Brau... male     22     1     0 ...
##  2           2        1      1 Cumi... female   38     1     0 ...
##  3           3        1      3 Heik... female   26     0     0 ...
##  4           4        1      1 Futr... female   35     1     0 ...
##  5           5        0      3 Alle... male     35     0     0 ...
##  6           6        0      3 Mora... male    NaN     0     0 ...
##  7           7        0      1 McCa... male     54     0     0 ...
##  8           8        0      3 Pals... male      2     3     1 ...
##  9           9        1      3 John... female   27     0     2 ...
## 10          10        1      2 Nass... female   14     1     0 ...
## # ... with more rows, and 1 more variable: Embarked <chr>


# More datasets can be stored in the same connection:
cars_tbl <- copy_to(sc, mtcars)

# List the available tables:
src_tbls(sc)
## [1] "mtcars"        "titanic_train"
```

Using dplyr Functins in sparklyr

The functionality that we are familiar with from dplyr can also be applied on the Spark table via sparklyr. The following code demonstrates this by summarizing some aspects of the data.

```
Titanic_tbl %>% summarise(n = n())
## # Source: spark<?> [?? x 1]
##       n
##   <dbl>
## 1   891

# Alternatively:
Titanic_tbl %>% spark_dataframe() %>% invoke("count")
## [1] 891

Titanic_tbl                                    %>%
  dplyr::group_by(Sex, Embarked)               %>%
  summarise(count = n(), AgeMean = mean(Age)) %>%
  collect
## # A tibble: 7 x 4
##   Sex    Embarked count AgeMean
##   <chr>  <chr>    <dbl>   <dbl>
## 1 male   C           95    33.0
## 2 male   S          441    30.3
## 3 female C           73    28.3
## 4 female ""           2    50
## 5 female S          203    27.8
## 6 male   Q           41    30.9
## 7 female Q           36    24.3
```

User Defined Functions with Spark_apply

```
# sdf_len creates a DataFrame of a given length (5 in the example)
x <- sdf_len(sc, 5, repartition = 1) %>%
     spark_apply(function(x) pi * x^2)
print(x)
## # Source: spark<?> [?? x 1]
##       id
##    <dbl>
## 1  3.14
## 2 12.6
## 3 28.3
## 4 50.3
## 5 78.5
```

The library `sparklyr` comes with a wide range of machine learning functions as well as with transforming functions.

- **feature transformers** are the functions that create buckets, binary values, element-wise products, binned categorical values based on quantiles, etc. The name of these functions starts with `ft_`. Note especially the function `sql_transformer()` – an exception on the naming convention – it is the function that allows to transform data based on an SQL statement.
- **machine learning algorithms** are the functions that perform linear regression, PCA, logistic regression, decision trees, random forest, etc. The name of these functions starts with `ml_`.

Choosing While Both SparkR and sparklyr

- mimic RStudio's `dplyr`,
- should feel reasonably familiar (knowing both base R and the `tidyverse`),
- allow to filter data and aggregate Spark datasets,
- have a substantial machine learning library that links through to MLlib,
- allow the results to be brought back to R for further analysis, visualization and reporting.

Reasons to choose SparkR

- `SparkR` has a Python equivalent, so if you use both languages you will appreciate the shortened learning curve;
- `sparkR` follows closer the scala-spark API and so also here are learning synergies.

Reasons to choose sparklyr

- smoother integration with `dpylr`,
- function naming conventions that are in line with the tidyverse conventions, and
- probably easier to learn for the modern R-user.

part: Bigger and Faster R

↓

chapter:

The Need for Speed

part: Bigger and Faster R

↓

chapter: The Need for Speed

↓

section:

Benchmarking

```
N <- 1500

# Load microbenchmark:
library(microbenchmark)

# Create a microbenchmark object:
comp <- microbenchmark(mean(x),              # 1st code block
                       {sum(x) / length(x)}, # 2nd code block
                       times = N)            # number times to run
```

The object `comp` is an object of class `microbenchmark` and – because of R's useful implementation of the S3 OO system – we can print it or ask for a summary with the known functions `print()` and `plot()` for example.

```
summary(comp)
##                       expr   min     lq     mean median     uq    max
## 1                  mean(x) 9.029 9.2235 9.359903 9.2885 9.3855 31.045
## 2 {     sum(x)/length(x) } 9.632 9.7180 9.822847 9.7650 9.8080 36.909
##   neval
## 1  1500
## 2  1500
```

Plotting the `microbenchmark` Object with `autoplot()`

```
# Load ggplot2:
library(ggplot2)

# Use autoplot():
autoplot(comp)
```

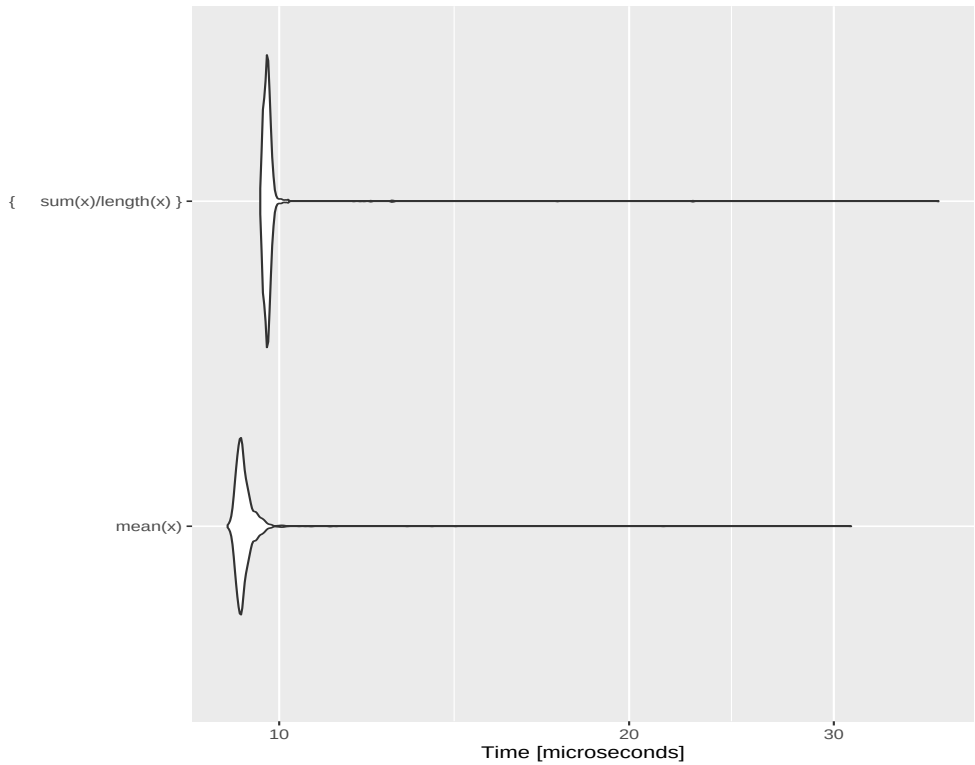

Figure 5: The package `microbenchmark` also defines a suible visualisation via `autoplot`. This shows us the violin plots of the different competing methods. It also makes clear how heavy the right tails of the distribution of run-times are.

part: Bigger and Faster R

↓

chapter: The Need for Speed

↓

section:

Optimize Code

Avoid Repeating the Same

Minimize repeating the same might sound obvious, but it will surprisingly often appear in early versions of code. For example, if you have a for-loop that needs in the length of a vector, then it is usually faster to do this once – before the loop – and store the result to be used in the for-loop.

```
x <- 1:1e+4
y <- 0

# Here we recalculate the sum multiple times.
system.time(for(i in x) {y <- y + sum(x)})
##    user  system elapsed
##   0.099   0.000   0.100

# Here we calculate it once and store it.
system.time({sum_x <- sum(x); for(i in x) {y <- y + sum_x}})
##    user  system elapsed
##   0.001   0.000   0.001
```

Use Vectorisation where Appropriate I

In R, vectors (and the family of `*apply`-functions) are usually faster than `for`-loops. This is true, but many tutorials on speed will make the bold statement that it is always better to use vectors than scalars. The following example, however, illustrates that there is a nuance to this.

```
# Define some numbers.
x1 <- 3.00e8
x2 <- 663e-34
x3 <- 2.718282
x4 <- 6.64e-11

y1 <- pi
y2 <- 2.718282
y3 <- 9.869604
y4 <- 1.772454

N <- 1e5
```

```
# 1. Adding some of them directly:
f1 <- function () {
  for (i in 1:N) {
    x1 + y1
    x2 + y2
    x3 + y3
    x4 + y4
    }
  }
system.time(f1())
##    user  system elapsed
##   0.027   0.000   0.027
```

Use Vectorisation where Appropriate II

```
# 2. Converting first to a vector and then adding the vectors:
f2 <- function () {
  x <- c(x1, x2, x3, x4)
  y <- c(y1, y2, y3, y4)
  for (i in 1:N) x + y
}
system.time(f2())
##    user  system elapsed
##   0.009   0.000   0.008
```

```
# 3. Working with the elements of the vectors:
f3 <- function () {
  x <- c(x1, x2, x3, x4)
  y <- c(y1, y2, y3, y4)
  for (i in 1:N) {
    x[1] + y[1]
    x[2] + y[2]
    x[3] + y[3]
    x[4] + y[4]
  }
  }
system.time(f3())
##    user  system elapsed
##   0.017   0.000   0.018
```

Use Vectorisation where Appropriate III

```
# 4. Working with the elements of the vectors and code shorter:
f4 <- function () {
  x <- c(x1, x2, x3, x4)
  y <- c(y1, y2, y3, y4)
  for (i in 1:N) for(n in 1:4) x[n] + y[n]
  }
system.time(f4())
##    user  system elapsed
##   0.024   0.000   0.024
```

Pre-allocating Memory I

Allocating memory takes a small time. While this is of the order of nanoseconds, it becomes significant if that is repeated enough. This is especially true for more complex objects. The preferable way to go is allocating the empty object first and then adding elements.

```
N <- 2e4

# Method 1: using append():
system.time ({
    lst <- list()
    for(i in 1:N) {lst <- append(lst, pi)}
   })
##    user  system elapsed
##   1.106   0.004   1.111
```

```
# Method 2: increasing length while counting with length():
system.time ({
    lst <- list()
    for(i in 1:N) {
        lst[[length(lst) + 1]] <- pi}
   })
##    user  system elapsed
##    0.01    0.00    0.01
```

Pre-allocating Memory II

```
# Method 3: increasing length using a counter:
system.time ({
    lst <- list()
    for(i in 1:N) {lst[[i]] <- pi}
   })
##    user  system elapsed
##   0.009   0.000   0.009
```

```
# Method 4: pre-allocate memory:
system.time({
    lst <- vector("list", N)
    for(i in 1:N) {lst[[i]] <- pi}
    })
##    user  system elapsed
##   0.006   0.000   0.005
```

Use the Most Simple Data Structure

Whenever you have the choice between a complex data-structure (RC object, list, etc.) and there is a more simple data-structure available – that can be used in the particular code – then choose for the most simple one. When an object provides more flexibility (e.g. a data-frame can also hold character types, where a matrix can only have elements of the same type), then this additional flexibility comes at a cost.
For example, let us investigate the difference between using a matrix and a data frame. Below we will compare both for simple arithmetic.

```
N <- 500
# simple operations on a matrix
M <- matrix(1:36, nrow = 6)
system.time({for (i in 1:N) {x1 <- t(M); x2 <- M + pi}})
##    user  system elapsed
##   0.003   0.000   0.003

# simple operations on a data-frame
D <- as.data.frame(M)
system.time({for (i in 1:N) {x1 <- t(D); x2 <- D + pi}})
##    user  system elapsed
##   0.166   0.000   0.166
```

Obviously, the aforementioned code does nothing useful: it only has unnecessary repetitions, but it illustrate that the performance gain of a matrix over a data-frame is massive (the matrix is more than 30 times faster in this example).

Use the Fastest Function

Functions in base R and larger projects such as the tidyverse are very fast and it will for the casual user not be possible to create a function with the same functionality that runs faster. Often these functions are written in C or C++ and are complied for speed. There is, however, a caveat. That caveat is in the words “same functionality.” Base functions such as `mean()` do a lot more than just `sum(x)/length(x)`. These base-functions are dispatcher functions: they will check the data-type of x and then dispatch to the correct function to calculate the mean.
We refer to the example from the beginning of this chapter:

```
x <- 1:1e4
N <- 1000
system.time({for (i in 1:N) mean(x)})
##    user  system elapsed
##   0.011   0.000   0.011

system.time({for (i in 1:N) sum(x) / length(x)})
##    user  system elapsed
##   0.012   0.000   0.011
```

Use the Fastest Package

```
N <- 732
# Use ts from stats to create the a time series object:
t1 <- stats::ts(rnorm(N), start = c(2000,1), end = c(2060,12),
                frequency = 12)
t2 <- stats::ts(rnorm(N), start = c(2010,1), end = c(2050,12),
                frequency = 12)

# Create matching zoo and xts objects:
zoo_t1 <- zoo::zoo(t1)
zoo_t2 <- zoo::zoo(t2)
xts_t1 <- xts::as.xts(t1)
xts_t2 <- xts::as.xts(t2)

# Run a merge on them:
# Note that base::merge() is a dispatcher function.
system.time({zoo_t <- merge(zoo_t1, zoo_t2)})
##    user  system elapsed
##   0.026   0.000   0.026

system.time({xts_t <- merge(xts_t1, xts_t2)})
##    user  system elapsed
##   0.001   0.000   0.001

# Calculate the lags:
system.time({for(i in 1:100) lag(zoo_t1)})
##    user  system elapsed
##   0.019   0.000   0.018

system.time({for(i in 1:100) lag(xts_t1)})
##    user  system elapsed
##   0.043   0.000   0.043
```

Details I

Did you think that curly brackets are as fast as round brackets? Or did you think that raising something to the power -1 is the same as division? Well, mathematically, these are the same things, but in a programming language such as R, they are typically executed in a different way and display differences in performance.

Note – Cold code ahead

Till now the differences in performance were both obvious and significant. For the remainder of the chapter, we study some effects that can be smaller and more subtle. Therefore, we have chosen to not to run the calculations and performance tests separate from compiling the book. In other words, while in the rest of the book – almost everywhere – the code that you see is directly generating the output and plots, below the code is run separately and results were manually added. The reason is that while generating this book, the code will be wrapped in other functions such as `knitr()`, `tryCatch()`, etc. and that this has a less predictable impact.
You can recognize the static code on the response of R: the output lines – starting with `##` – appear in bold green and not in black text.

Details II

Consider the following alternatives to calculate $\frac{1}{1+x}$.

```
# standard function:
f1 <- function(n, x = pi) for(i in 1:n) x = 1 / (1+x)

# using curly brackets:
f2 <- function(n, x = pi) for(i in 1:n) x = 1 / {1+x}

# adding unnecessary round brackets:
f3 <- function(n, x = pi) for(i in 1:n) x = (1 / (1+x))

# adding unnecessary curly brackets:
f4 <- function(n, x = pi) for(i in 1:n) x = {1 / {1+x}}

# achieving the same result by raising to a power
f5 <- function(n, x = pi) for(i in 1:n) x = (1+x)^(-1)

# performing the power with curly brackets
f6 <- function(n, x = pi) for(i in 1:n) x = {1+x}^{-1}

N <- 1e6
library(microbenchmark)
comp <- microbenchmark(f1(N), f2(N), f3(N), f4(N), f5(N), f6(N),
                       times = 150)

comp
## Unit: milliseconds
##   expr      min       lq     mean   median       uq      max  ...
##   f1(N) 37.37476 37.49228 37.76950 37.57212 37.79876 39.99120 ...
##   f2(N) 37.29297 37.50435 37.79612 37.63191 37.81497 41.09414 ...
##   f3(N) 37.96886 38.18751 38.59619 38.28713 38.68162 47.66612 ...
##   f4(N) 37.88111 38.06787 38.41134 38.16297 38.36706 42.53103 ...
##   f5(N) 45.12742 45.31632 45.67364 45.45465 45.69882 49.65297 ...
##   f6(N) 45.93406 46.03159 46.51151 46.15287 46.64509 52.95426 ...
```

Compile Functions I

Depending on the particular code, compiling R-code can be a major time gain. Core-R offers the library `compiler` to achieve this in a surprisingly easy and straightforward way.

```
library(compiler)
N <- as.double(1:1e7)

# Create a *bad* function to calculate mean:
f <- function(x) {
  xx = 0
  l = length(x)
  for(i in 1:l)
    xx = xx + x[i]/l
  xx
}

# Time the function:
system.time(f(N))
##   user  system elapsed
##  0.61    0.000    0.61

# Compile the function:
cmp_f <- cmpfun(f)

# Time the compiled version
system.time(cmp_f(N))
##  user  system elapsed
##  0.596  0.00    0.596
```

Compile Functions II

```
# The difference is small, so we use microbenchmark
library(microbenchmark)
comp <- microbenchmark(f(N), cmp_f(N), times = 150)

# See the results:
comp
## Unit: milliseconds
##      expr      min       lq     mean   median       uq      ...
##      f(N) 552.2785 553.9911 559.6025 556.1511 562.7207 601.5...
##  cmp_f(N) 552.2152 553.9453 558.1029 555.8457 560.0771 588.4...


# Plot the results.
library(ggplot2)
autoplot(comp)
```

Compile Functions III

The plot, generated by the aforementioned code is in Figure 6 on slide 94. The compiled function is just a little faster, but shows a more consistent result.

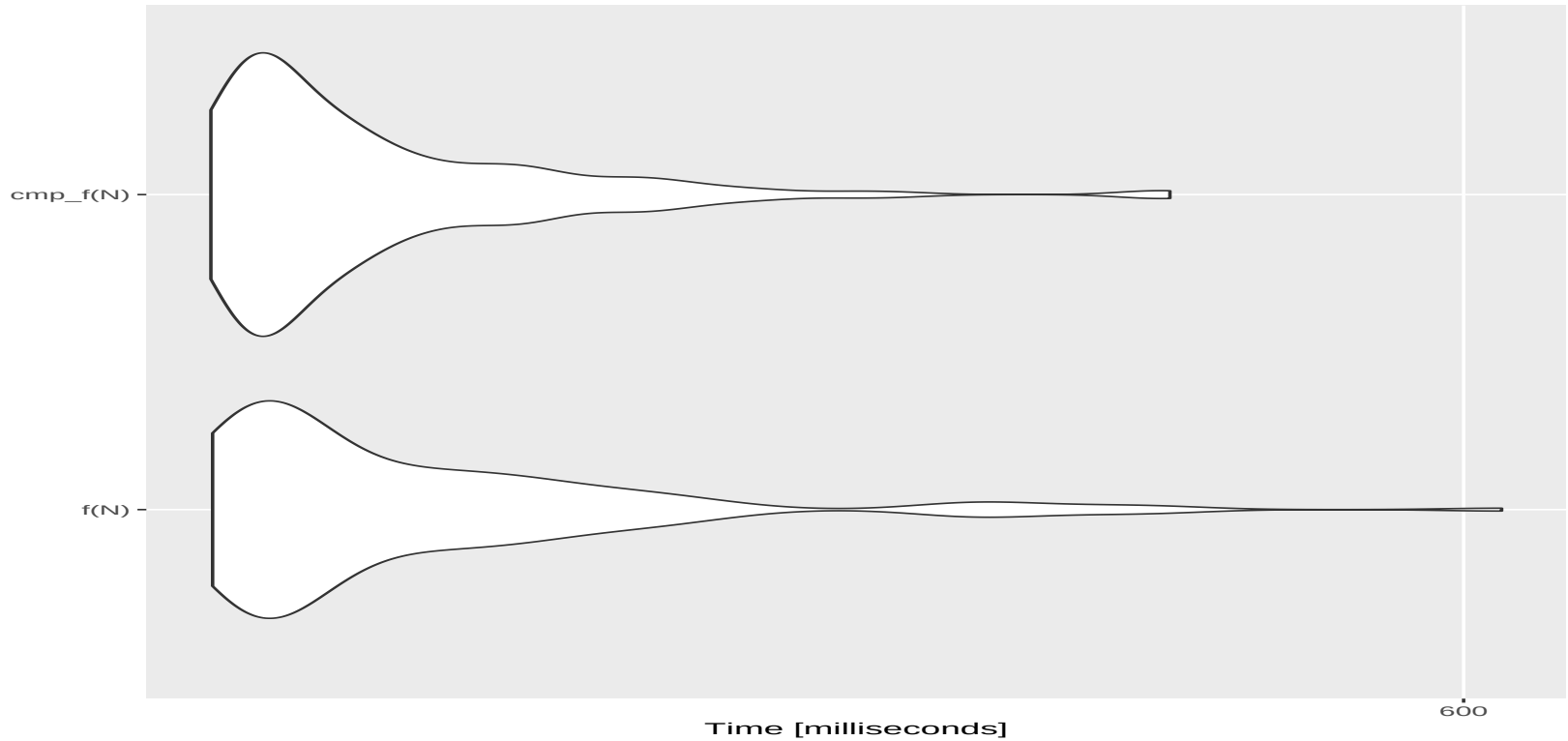

Figure 6: Autoplot generates nice violin plots to compare the speed of the function defined in the aforementioned code (f) versus its compiled version (cmp_f).

Use C or C++ Code in R I

To illustrate how simple is is to use C++ code via Rcpp we will create a function to calculate the Fibonacci numbers. The Fibonacci numbers are so that $F_n = F_{n-2} + F_{n-1}$ and $F_0 = F_1 = 1$. This inspires us to write the following naive and inefficient function:

```
# Naive implementation of the Fibonacci numbers in R:
Fib_R <- function (n) {
  if ((n == 0) | (n == 1)) return(1)
  return (Fib_R(n - 1) + Fib_R(n - 2))
  }

# The R-function compiled via cmpfun():
library(compiler)
Fib_R_cmp <- cmpfun(Fib_R)

# Using native C++ via cppFunction():
Rcpp::cppFunction('int Fib_Cpp(int n) {
  if ((n == 0) || (n == 1)) return(1);
  return (Fib_Cpp(n - 1) + Fib_Cpp(n - 2));
}')

library(microbenchmark)
N <- 30
comp <- microbenchmark(Fib_R(N), Fib_R_cmp(N),
                       Fib_Cpp(N), times = 25)

comp
## Unit: milliseconds
##          expr         min          lq        mean      median          uq
##      Fib_R(N) 1449.755022 1453.320560 1474.679341 1456.202559 1472.447928
##  Fib_R_cmp(N) 1444.145773 1454.127022 1489.742750 1459.170600 1554.450501
##    Fib_Cpp(N)    2.678766    2.694425    2.729571    2.711567    2.749208
##          max neval cld
##  1596.226483    25   b
```

The last line of the aforementioned code generates a plot, that is in Figure 7 on slide 96.

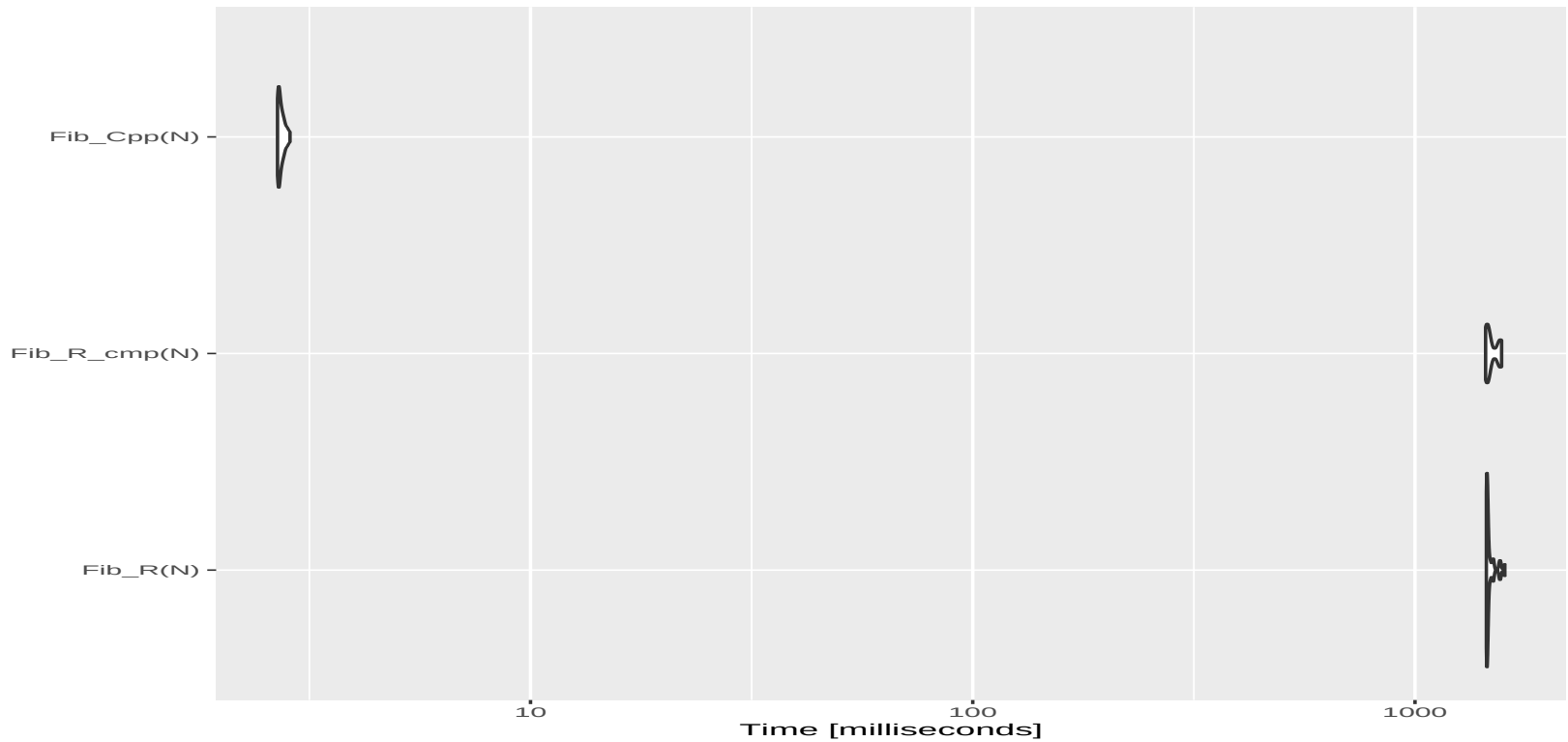

FIGURE 7: A violin plot visualizing the advantage of using C++ code in R via `Rcpp`. This picture makes very clear that the "compilation" (via the function `cmpfun()`) has only a minor advantage over a native R-function. Using C++ and compiling via a regular compiler (in the background `cppFunction()` uses g++), however, is really a game changer.

Use C or C++ Code in R ...and Use a Smart Approach I

```r
# Efficient function to calculate the Fibonacci numbers in R:
Fib_R2 <- function (n) {
 x = 1
 x_prev = 1
 for (i in 2:n) {
   x <- x + x_prev
   x_prev = x
   }
 x
 }

# Efficient function to calculate the Fibonacci numbers in C++:
Rcpp::cppFunction('int Fib_Cpp2(int n) {
  int x = 1, x_prev = 1, i;
  for (i = 2; i <= n; i++) {
    x += x_prev;
    x_prev = x;
    }
  return x;
  }')
```

Use C or C++ Code in R ...and Use a Smart Approach II

```
# Test the performance of all the functions:
N <- 30
comp <- microbenchmark(Fib_R(N), Fib_R2(N),
                       Fib_Cpp(N), Fib_Cpp2(N),
                       times = 20)

comp
## Unit: microseconds
##         expr         min           lq         mean     median ...
##     Fib_R(N) 1453850.637 1460021.5865 1.495407e+06 1471455.852...
##    Fib_R2(N)       2.057       2.4185 1.508404e+02       4.792...
##   Fib_Cpp(N)    2677.347    2691.5255 2.757781e+03    2697.519...
##  Fib_Cpp2(N)       1.067       1.4405 5.209175e+01       2.622...
##          max neval cld
##  1603991.462    20   b
##     2925.070    20  a
##     3322.077    20  a
##      964.378    20  a

library(ggplot2)
autoplot(comp)
```

Use C or C++ Code in R ...and Use a Smart Approach III

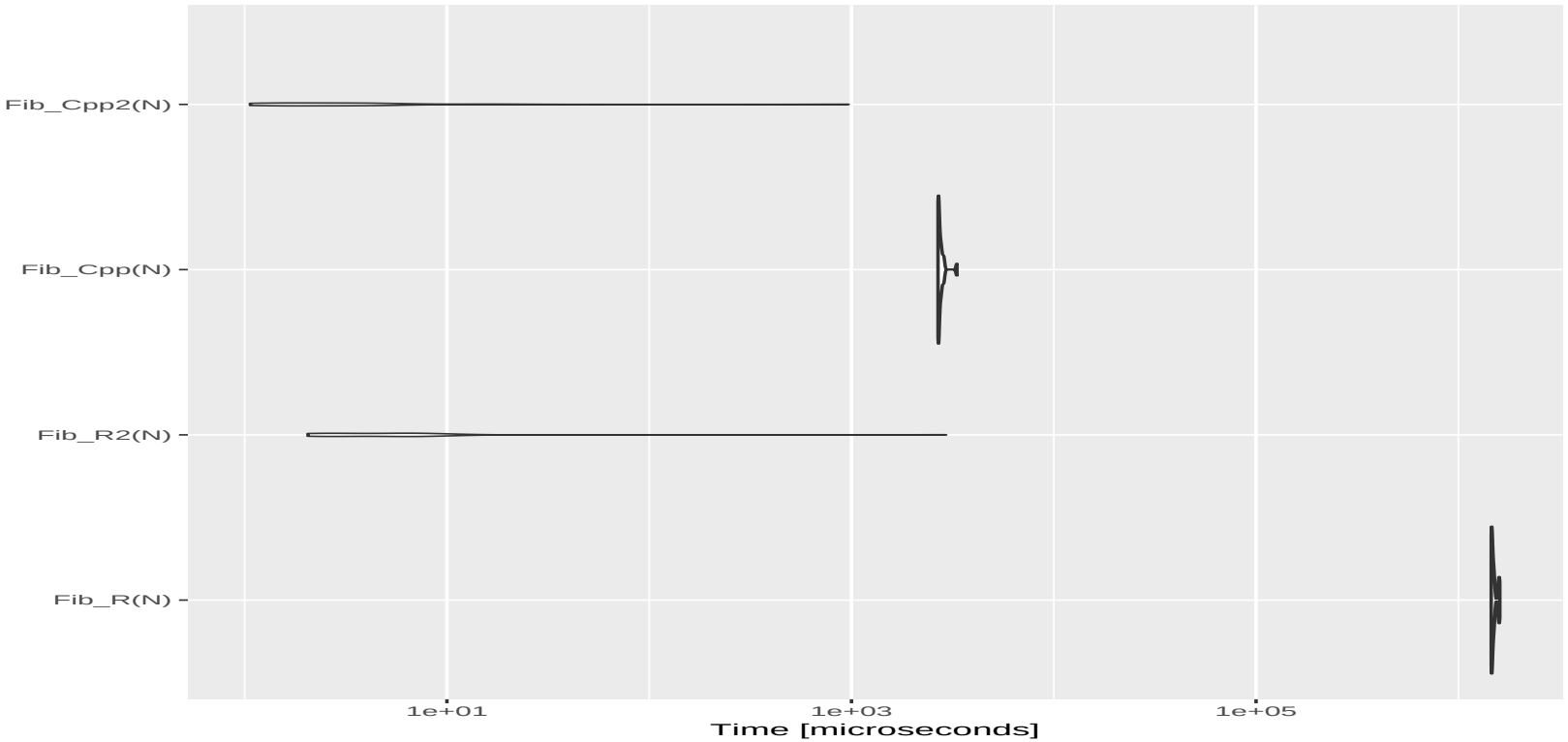

Figure 8: A violin plot visualizing the advantage of using C++ code in R via Rcpp. Note that the scale of the *x*-axisdefaulted to a logarithmic one. This image makes very clear that using native C++ via Rcpp is a good choice, but that it is equally important to be smart.

`cppFunction()` uses a string as argument and while this works fine for short functions, it becomes unwieldy for large chunks of code. In fact, we might want to include header files or simply define more than one function. In such case, we can include C++ source files and via `sourceCpp()`. This allows us also to use a regular C++ IDE or text editor, with C++ syntax highlighting and line numbers, save time programming and debugging, while it becomes also a smoother experience.
To achieve the integration of the C++ code in R, it is sufficient to abide to the following rules. First, the C++ source file should have the usual `.cpp` extension, and needs to start with:

```
#include <Rcpp.h>
using namespace Rcpp;
```

Further, the functions that needs to be exported to R should be preceded with exactly the following comment (nothing more and nothing less in the line preceding the function – not even an empty line):

```
// [[Rcpp::export]]
```

Hint – R code within the C++ code within R

Note also that it is possible to embed R code in special C++ comment blocks. This is really convenient if you want to run some test code, or want to keep functions logically in the same place (regardless the programming language used):

```
/*** R
# This is R code
*/
```

The R code is run with `source(echo = TRUE)` so you don't need to explicitly print output.

Once this source file is created and saved on the hard disk we can compile it in R via `sourceCpp()`.
For further reference we will assume that the file is saved under the name `/path/cppSource.cpp`.

```
sourceCpp("/path/cppSource.cpp")
```

The function `sourceCpp()` will create all the selected functions and make them available to R in the current environment.

Warning – C++ function not saved in .RData

As you know R is able to remember the current environment via saving all variables and functions in the `.Rdata` file, and then load all that information when a new session is opened. Note that the C++ functions will not be saved in and must be created again in a future session.

In the following example we use the example from previous section: the faster implementations of the Fibonacci series in both C++ and R. To import this function from C++ source, we need to create a file with the following content:

Using a C++ Source File in R IV

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int Fib_Cpp2(int n) {
  int x = 1, x_prev = 1, i;
  for (i = 2; i <= n; i++) {
    x += x_prev;
    x_prev = x;
    }
  return x;
  }

/*** R
Fib_R2 <- function (n) {
 x = 1
 x_prev = 1
 for (i in 2:n) {
   x <- x + x_prev
   x_prev = x
   }
 x
 }


N <- 30
comp <- microbenchmark(Fib_R2(N), Fib_Cpp2(N), times = 20)

comp
*/
```

This source file will not only make the C++ function available but also run some tests, so that we can immediately assess if all works fine. When this file is imported via the function `sourceCpp()`, the functions `Fib_Cpp2()` and `Fib_R()` will both be available to be used in R.

Existing (compiled C and C++) functions – that sit on the hard-drive of the computer as binaries – can be called from the R environment via `.Call()` function.
R objects passed to these routines have type `SEXP`. These objects are pointers to a structure that holds the object's type, value, and other attributes. The R application programming interface (API) provides a limited set of functions to call these.

Further information – Calling C functions

More information about the subject of calling C routines in R, is for example here: `http://adv-r.had.co.nz/C-interface.html`

part: Bigger and Faster R

↓

chapter: The Need for Speed

↓

section:

Profiling code

Definition 1 (Profiling)

Profiling is the process of finding out what part of the code takes most time to run is called “profiling

Note – Cold code ahead

While in the rest of the book the code shown generates directly the output and plots that appear below or near to it, the code in this section is run separately and results were manually collated. The reason is that while generating this book, the code will be wrapped in other functions such as `knitr()`, `tryCatch()`, etc. and that this has a less predictable impact and – most importantly – those functions might also appear in the output that we present here and this would be confusing for the reader.

Rprof from utils Is Your Workhorse

The package `utils` that is part of base-R provides the function `Rpof()` that is the profiling tool in R. The function is called when the logging should start with the name of the file where the results should be stored. Then follows the code to be profiled and finally we call `Rprof()` again to stop the logging. So, the general work-flow is as follows.

```
Rprof("/path/to/my/logfile")
... code goes here
Rprof(NULL)
```

The Example For Profiling

As a first example we will consider a simple construction of functions that mainly call each other, in such way that some parts should display a predictable performance difference (added complexity or double amount of repetitions).

```
f0 <- function() x <- pi^2
f1 <- function() x <- pi^2 + exp(pi)
f2 <- function(n) for (i in 1:n) {f0(); f1()}
f3 <- function(n) for (i in 1:(2*n)) {f0(); f1()}
f4 <- function(n) for (i in 1:n) {f2(n); f3(n)}
```

Running the Profiling

In order to do the profiling we have to mark the starting point of the profiling operation by calling the function `Rprof()`. This indicates to R that it should start monitoring. The function takes one argument: the file to keep the results. The profiling process will register information till the process is stopped via another call to `Rprof()` with the argument `NULL`.

```
# Start the profiling:
Rprof("prof_f4.txt")

# Run our functions:
N <- 500
f4(N)

# Stop the profiling process:
Rprof(NULL)
```

Analysing the Profiling Log

Now, the file `prof_f4.txt` is created and can be read and analysed. The function `SummaryRprof()` from the package `utils` is a first port of call and usually provides very good insight.

```
# show the summary:
summaryRprof("prof_f4.txt")
## $by.self
##      self.time self.pct total.time total.pct
## "f0"      0.18    37.50       0.18     37.50
## "f3"      0.16    33.33       0.34     70.83
## "f1"      0.08    16.67       0.08     16.67
## "f2"      0.06    12.50       0.14     29.17
##
## $by.total
##      total.time total.pct self.time self.pct
## "f4"       0.48    100.00      0.00     0.00
## "f3"       0.34     70.83      0.16    33.33
## "f0"       0.18     37.50      0.18    37.50
## "f2"       0.14     29.17      0.06    12.50
## "f1"       0.08     16.67      0.08    16.67
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 0.48
```

The times shown are:

- `by.self`: the time spent at the level of a certain function;
- `by.total`: the time spent in a function, including all the functions that this function calls.

The Visualisation with `profr` I

```
N <- 1000
require(profr)
pr <- profr({f4(N)}, 0.01)
plot(pr)
```

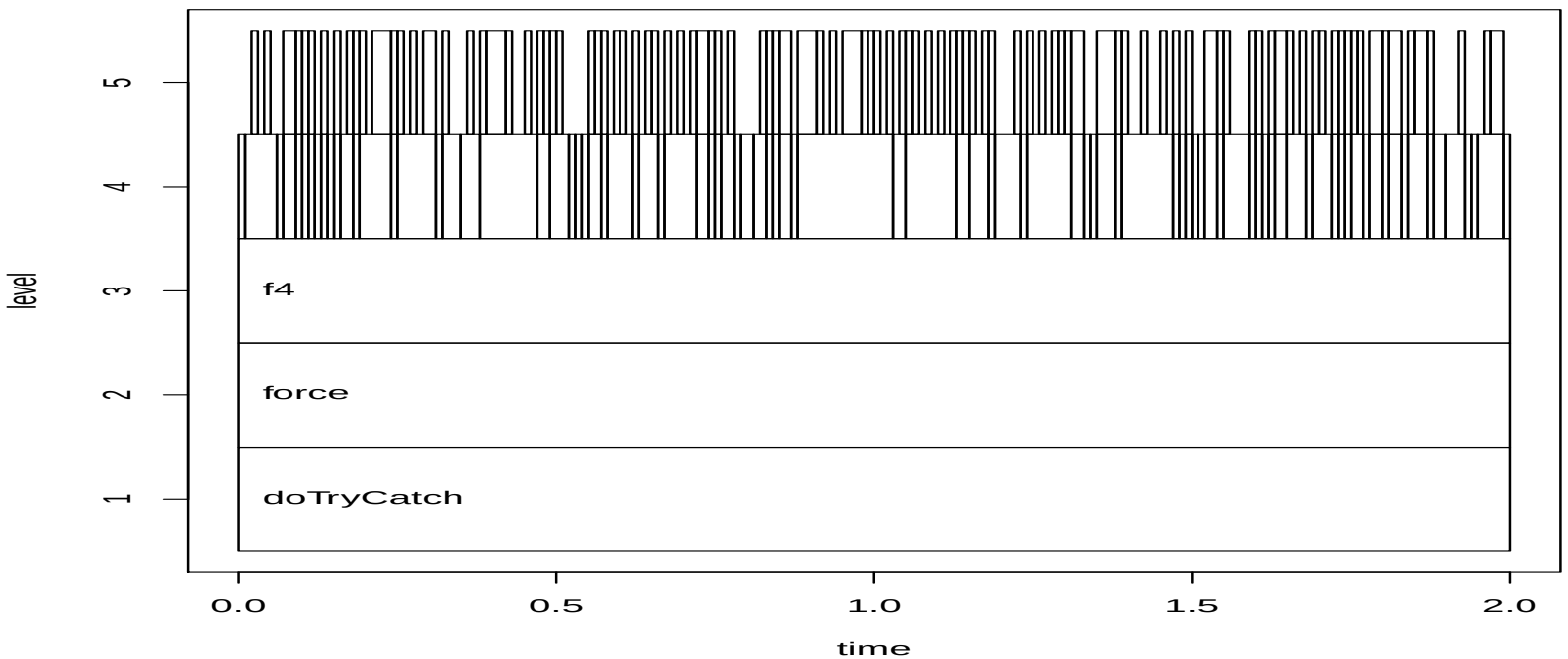

Figure 9: A visualization of the profiling of our functions.

The package `proftools` Allows More Visualisation I

```
library(proftools)

# Read in the existing profile data from Rprof:
pd <- readProfileData("prof_f4.txt")

# Print the hot-path:
hotPaths(pd, total.pct = 10.0)
##  path   total.pct self.pct
##  f4     100.00     0.00
##  . f3    70.83    33.33
##  . . f0  29.17    29.17
##  . f2    29.17    12.50

# A flame-graph (stacked time-bars: first following plot)
flameGraph(pd)
```

The package `proftools` Allows More Visualisation II

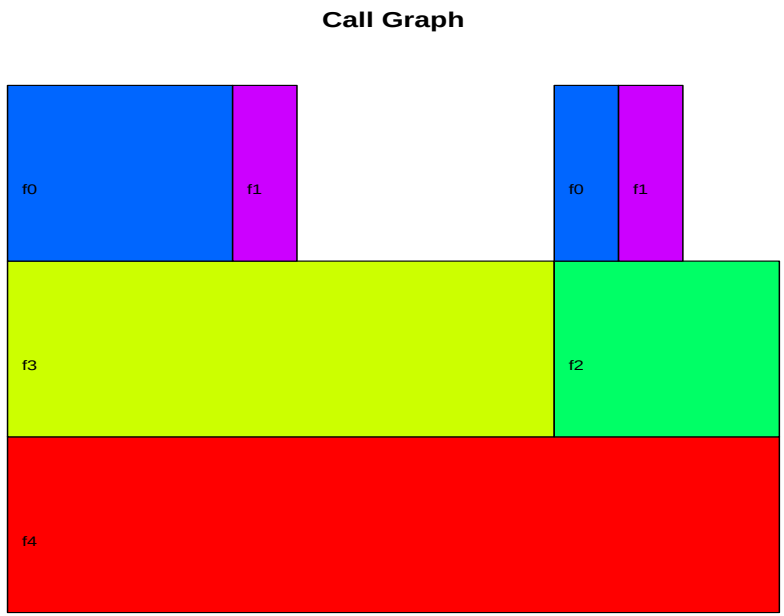

Figure 10: A flame-plot produced by the function `flameGraph()` from the package `proftools`.

```
# Callee tree-map (intersecting boxes with area related to time
# spent in a function: see plot below)
calleeTreeMap(pd)
```

The package `proftools` Allows More Visualisation III

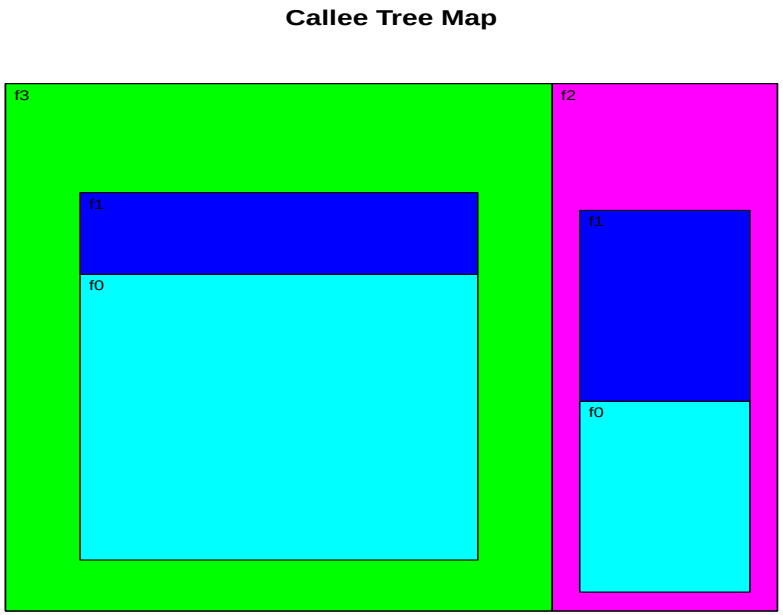

FIGURE 11: A Callee Tee Map produced by the function `calleeTreeMap()` from the package `proftools`. This visualization shows boxes with surfaces that are relative to the time that the function takes.

part: Bigger and Faster R

↓

chapter: The Need for Speed

↓

section:

Optimize Your Computer

Most efficiency gain can be obtainted from using a smart approach, fast functions, compiling code, using all equipment optimally, and efficient algorithms. We do not recommend tinkering with clock speed, niceness, etc. Gains will be marginal compared to the smart approach elaborated earlier in this chapter.