

Science Club – 20200909

R optimization tips & tricks (Benchmarked)

Deepankar Chakroborty
PhD student, Faculty of Medicine
University of Turku

Why optimize ?

- “Return on investment” for “Code Optimization” is +ve if:
 - The code is/will be re-used several times
 - It runs in a workflow / pipeline
 - You’re deploying it for end-users
 - The “vanilla” way takes too damn long
 - You want to learn more about the language.
- “Fast software is the best software” – Craig Mod
https://craigmod.com/essays/fast_software/

When not to optimize ?

- You’re on a deadline
- You’re just waiting for it to be over

Description of data

- Catalog of Somatic Mutations in Cancer COSMIC v91
 - <https://cancer.sanger.ac.uk/cosmic/download#download-3>
 - Complete size 16 GB (\approx 4G gzip RDS)
 - Processed to host 2 databases at:
<https://eleniuslabtools.utu.fi/main/HotspotExplorer.html>
- 40 columns x millions of rows (*literally!!*)
- For these benchmarks → subset files were used with:
 - 10, 100, 1 000, 10 000, 100 000, 1 000 000 rows each
 - Max file size: 350M

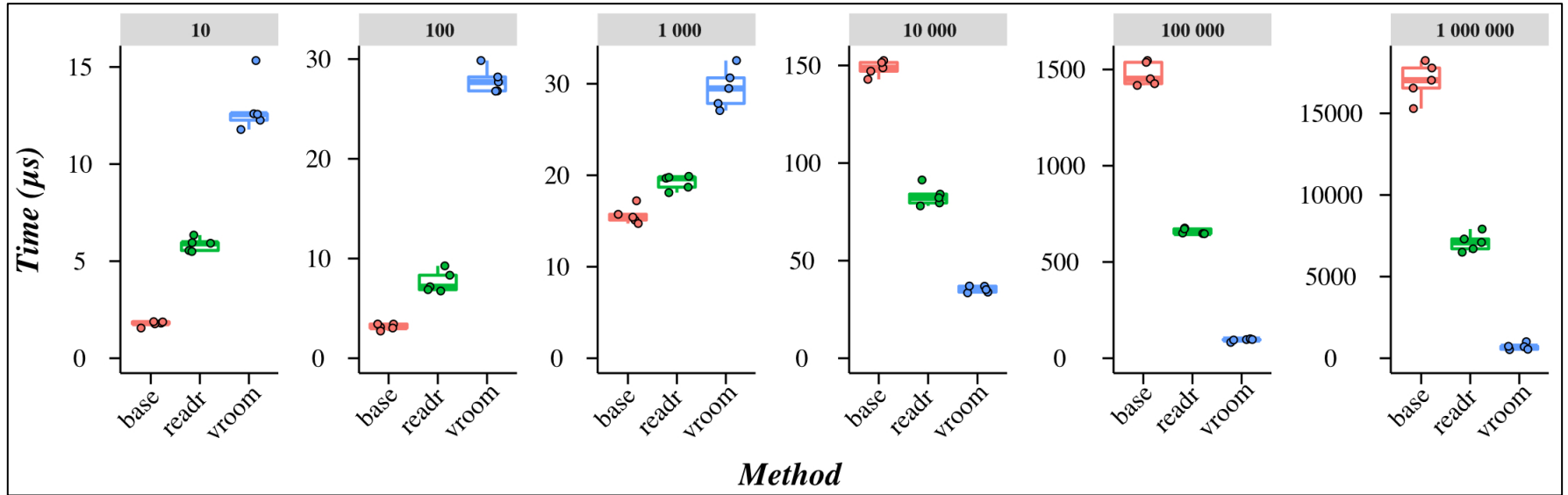
Test bench

- Tested on iMac running Mac OS Catalina
 - **Proc:** 3.2 GHz x 4 physical cores x 1 thread [i.e. *no SMT*]
 - **RAM** 24 GB 1600 MHz
- R version 3.6.3
 - vroom – 1.3.0
 - readr – 1.3.1
 - ggplot2 - 3.3.2
 - stringi - 1.4.6
 - parallel - 3.6.3
 - doParallel – 1.0.15
 - snow - 0.4-3
 - plyr – 1.8.6
 - microbenchmark – 1.4-7

Disclaimer !

- Tested on Mac OS (2 machines)
- Based on my experience
- Examples: some are *real*, **MOST** are *fabricated*
- I'm **NOT** here to tell you what is the “right way”.
Your way IS the right way for you.
I'm just offering alternative(s).
- These are definitely **NOT** the “fastest” way.
That would be to **write something in C**, etc.!

Reading files

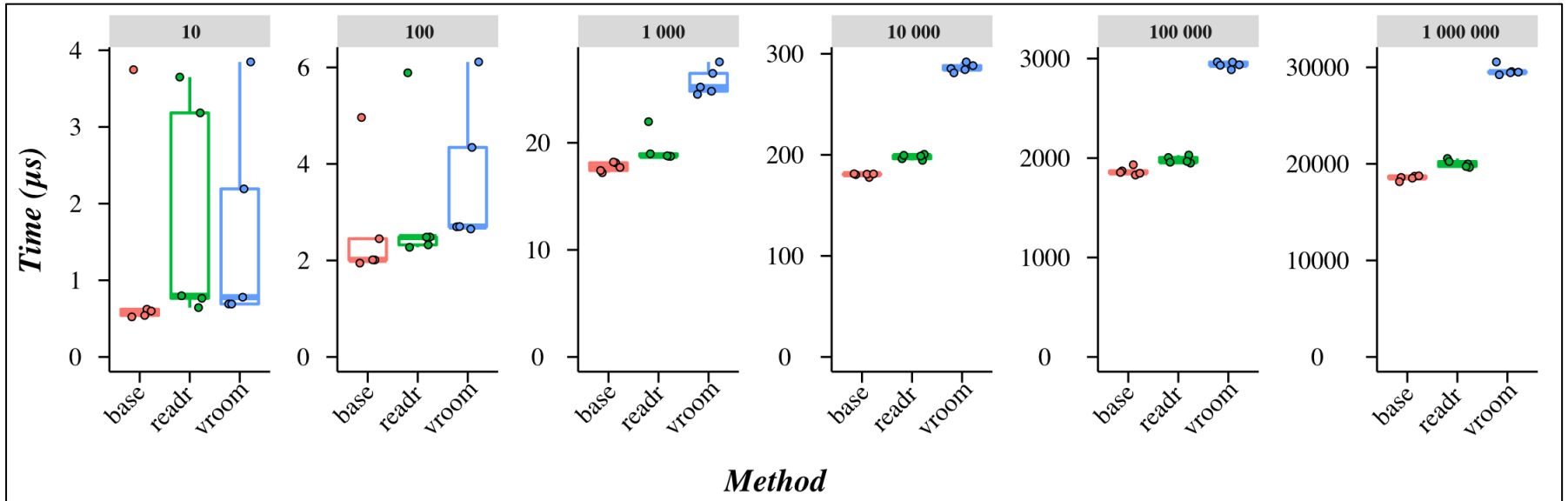
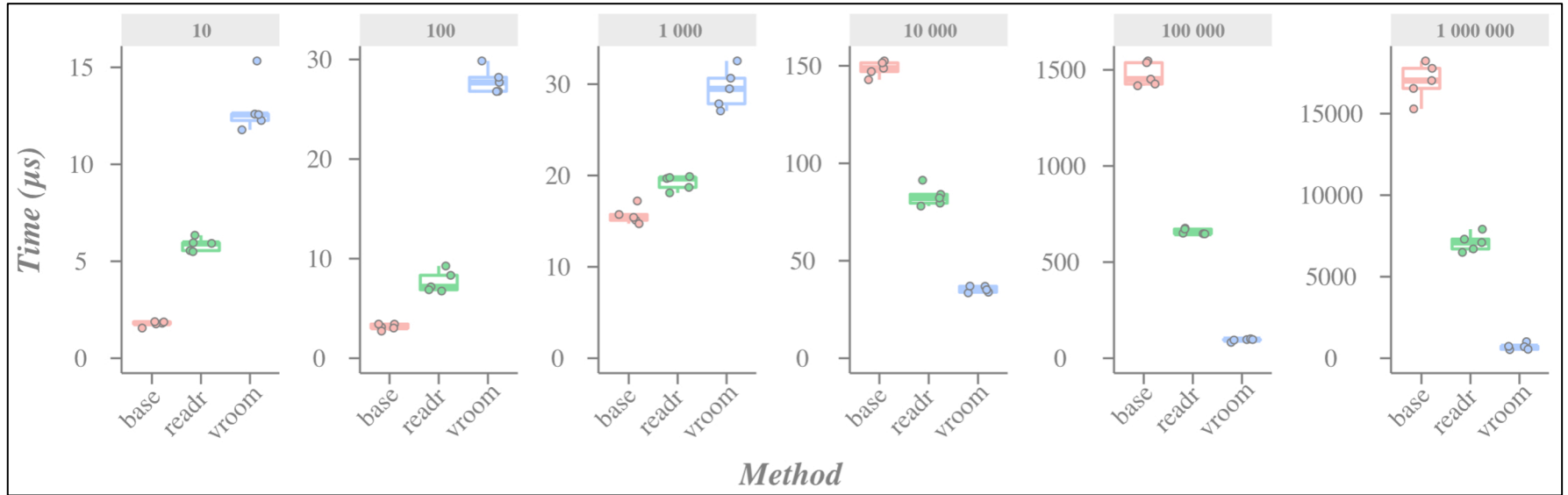


```
# base
dat <- utils::read.table(file = file.name, header = T, sep = "\t", as.is = T, stringsAsFactors = T)
; rm(dat)
```

```
-----
# readr
dat <- readr::read_delim(file = file.name, delim = "\t", progress=F) ; rm(dat)
```

```
-----
# vroom
dat <- vroom::vroom(file = file.name, delim = "\t", progress = F) ; rm(dat)
```

Reading files → Writing files



Populating a vector

```
strings <- dat[1:i,"HGVSC"] # ENST00000357360.4:c.*22C>T
```

```
# concat: concatenation
```

```
ENS_ID <- c()
for(idx in seq_along(strings)){
  ENS_ID <- c(ENS_ID, unlist(strsplit(x = strings[idx], split = ".", fixed = T),
use.names = F)[1])
}
```

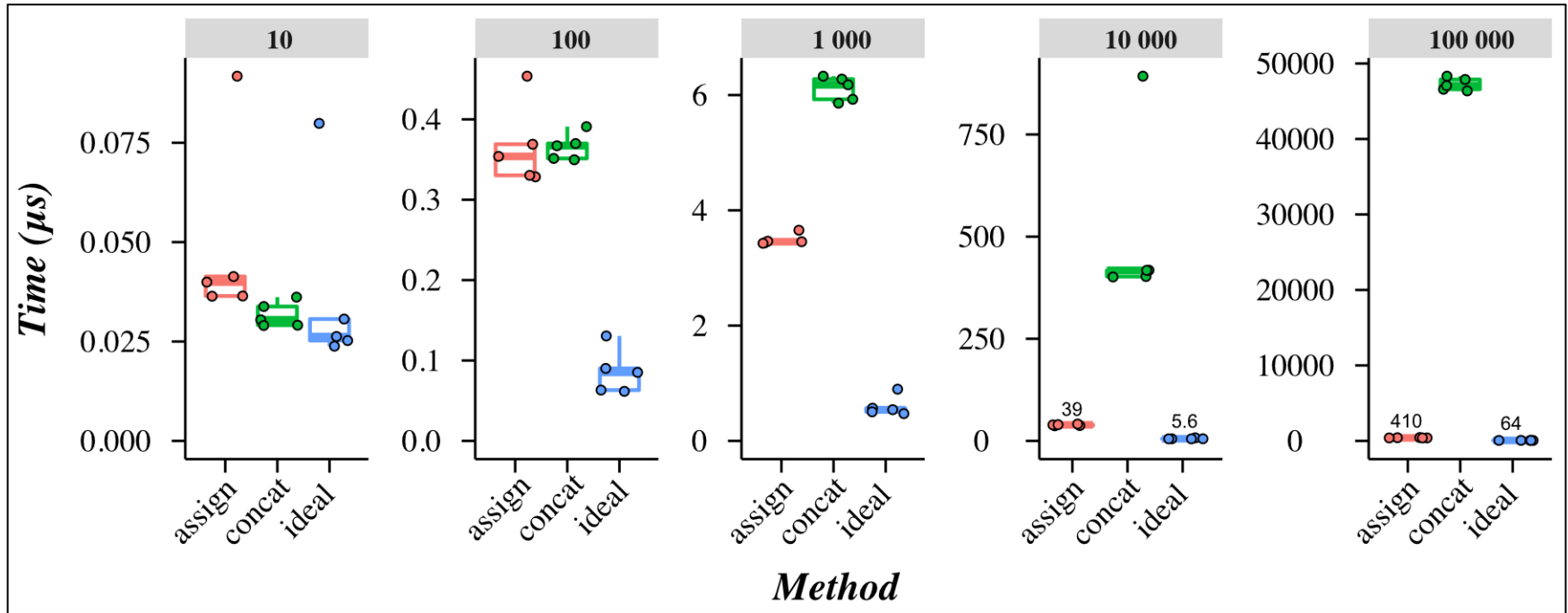
```
# assign
```

```
ENS_ID <- rep(NA,length(strings))
for(idx in seq_along(strings)){
  ENS_ID[idx] <- unlist(strsplit(x = strings[idx], split = ".", fixed = T),
use.names = F)[1]
}
```

```
# ideal: vectorization (i.e. skip the for loop)
```

```
ENS_ID <- unlist(strsplit(x = strings,split = ".", fixed = T), use.names = F)[seq(1,
3 * length(strings), by = 3)]
```


Populating a vector



```
# concat: concatenation
ENS_ID <- c() # declare vector
for(loop){
  ENS_ID <- c(ENS_ID, new_value)
}
```

```
# assign: assign to different indices
ENS_ID <- rep(NA,length(strings)) # Initialize an empty vector
for(loop){
  ENS_ID[index] <- new_value
}
```

```
# ideal: vectorization (i.e. skip the for loop)
ENS_ID <- unlist(strsplit(x = strings, split = ".", fixed = T), use.names = F)[seq(1, 3 * length(strings), by = 3)]
```

Vectorization

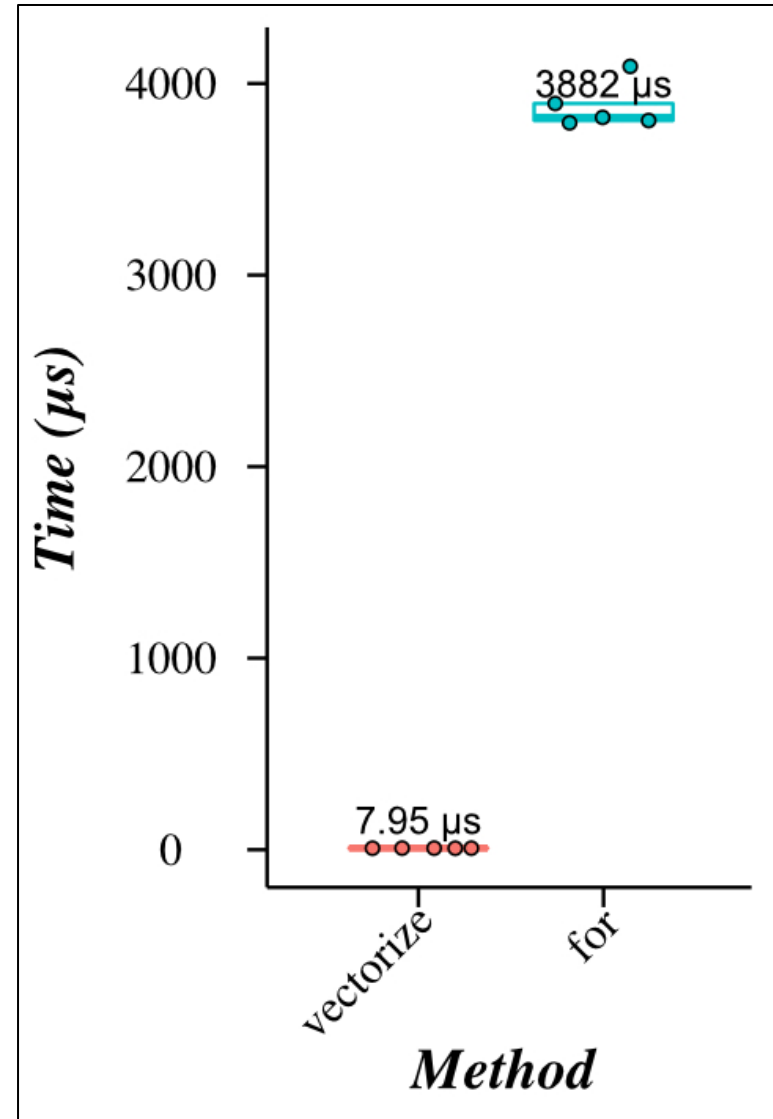
calculations with 3 numeric columns

vectorization

```
calc_vec <- ((dat$ID_tumour + dat$ID_sample)/  
dat$HGNC.ID)
```

for loop

```
calc_for <- rep(NA, length(dat$Gene.name))  
for(i in seq_along(dat$Gene.name)){  
  calc_for[i] <- ((dat$ID_tumour[i] +  
dat$ID_sample[i]) / dat$HGNC.ID[i])  
}
```



Vectorization, better example

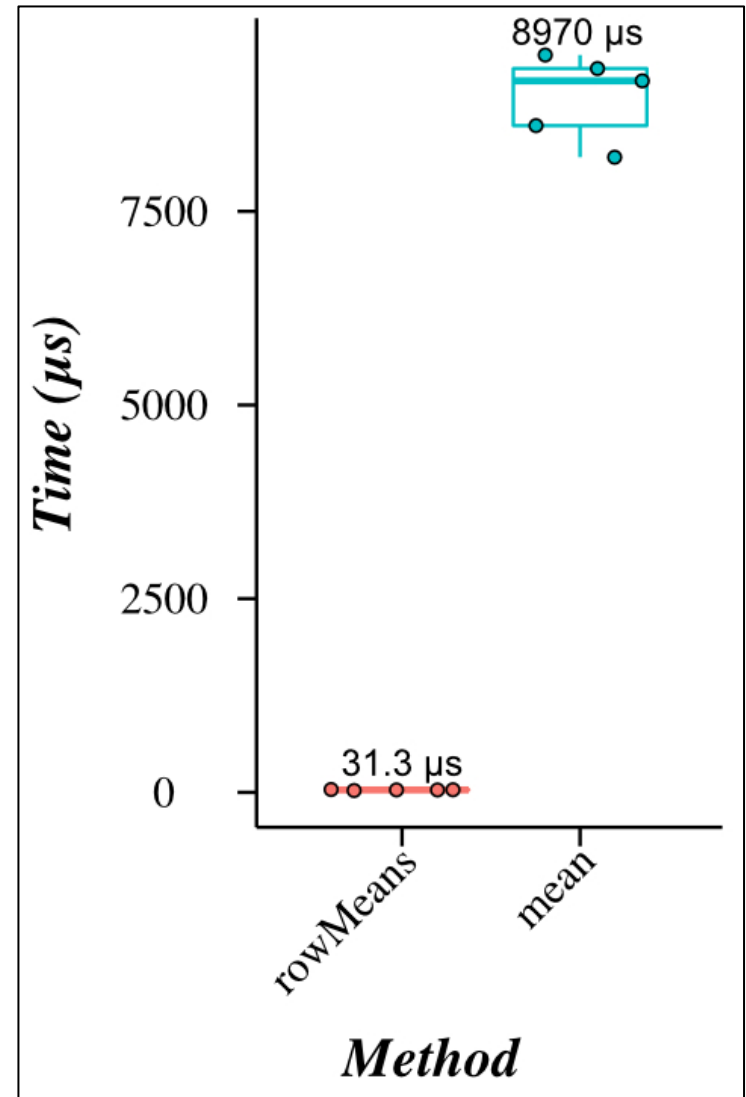
for each row calculate mean of value in 3 columns

vectorized

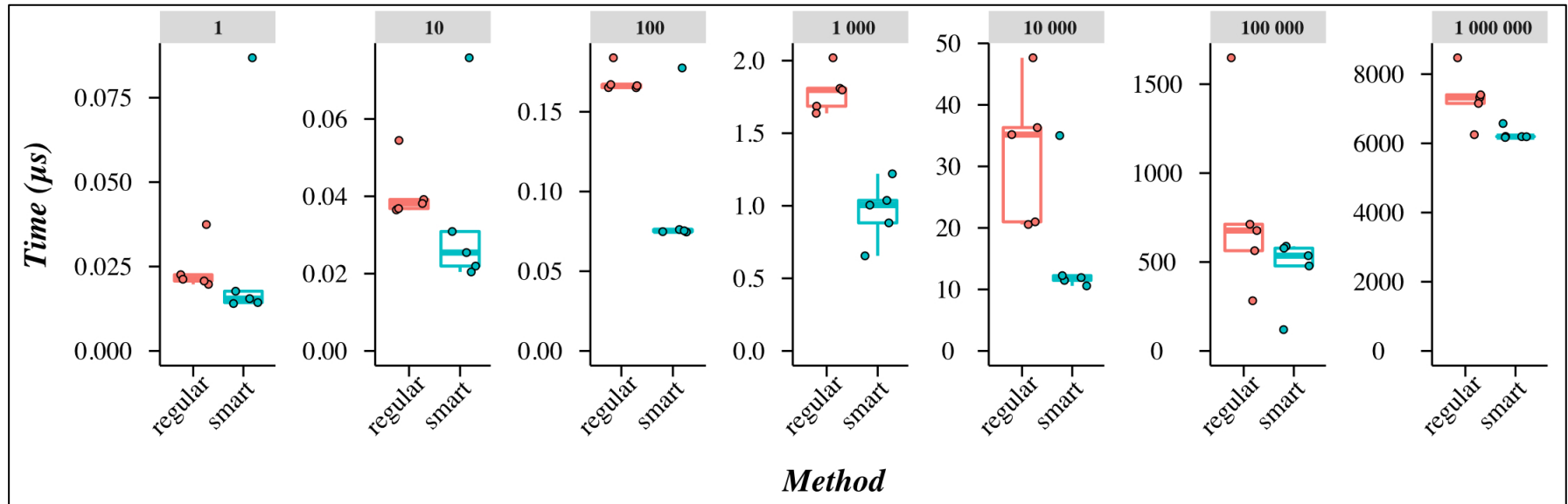
```
rM <- rowMeans(dat[,c("ID_sample", "ID_tumour",  
"MUTATION_ID")])  
,
```

for

```
M <- rep(NA, length(dat$Gene.name))  
for(i in seq_along(dat$Gene.name)){  
  M[i] <- ((dat$ID_tumour[i] + dat$ID_sample[i] +  
dat$MUTATION_ID[i]) / 3)  
}
```



Utilizing specific function parameters



"regular"

```
res_1 <- base::substring(text = genomePos, first = {unlist(base::gregexpr(pattern = ":", text = genomePos)) + 1})}
```

smart

```
res_2 <- base::substring(text = genomePos, first = {unlist(base::gregexpr(pattern = ":", text = genomePos, fixed = T), use.names = F) + 1})}
```

fixed=T, if you're not using regex [applicable in grep, gsub, etc.]

use.names, if you don't care about names of the values (or can add them later on as well).

for() vs apply()

```
# chr1:1200312511-1587831227" return -387518716
```

```
# for loop
```

```
res_for <- rep(NA,length(genomePosition))
for(i in seq_along(genomePosition)){
  genomePos <- base::substring(text = genomePosition[i],first = base::gregexpr(pattern = ":", text =
genomePosition[i], fixed = T)[[1]][1]+1)
  minus <- base::gregexpr(pattern = "-",text = genomePos, fixed = T)[[1]][1]
  res_for[i] <- as.integer(substring(text = genomePos,first = 1,last = minus-1))-as.integer(substring(text =
genomePos,first = minus+1))
}
```

```
# apply
```

```
find_length <- function(genomePos = NULL){
  genomePos <- base::substring(text = genomePos,first = base::gregexpr(pattern = ":",text = genomePos, fixed =
T)[[1]][1]+1)
  minus <- base::gregexpr(pattern = "-",text = genomePos, fixed = T)[[1]][1]
  return(as.integer(substring(text = genomePos,first = 1,last = minus-1))-as.integer(substring(text =
genomePos,first = minus+1)))
}
res_apply <- sapply(X = genomePosition,FUN = find_length,USE.NAMES = F)
```

```
# smart : define vectorized function that iterates by itself i.e. no need for apply
```

```
find_length_base <- function(genomePos = NULL){
  genomePos <- base::substring(text = genomePos,first = {unlist(base::gregexpr(pattern = ":",text =
genomePos, fixed = T),use.names = F)+1})
  minus <- unlist(base::gregexpr(pattern = "-",text = genomePos, fixed = T),use.names = F)
  return(as.integer(substring(text = genomePos,first = 1,last = minus-1))-as.integer(substring(text =
genomePos,first = minus+1)))
}
res_smart <- find_length_base(genomePos = genomePosition)
```

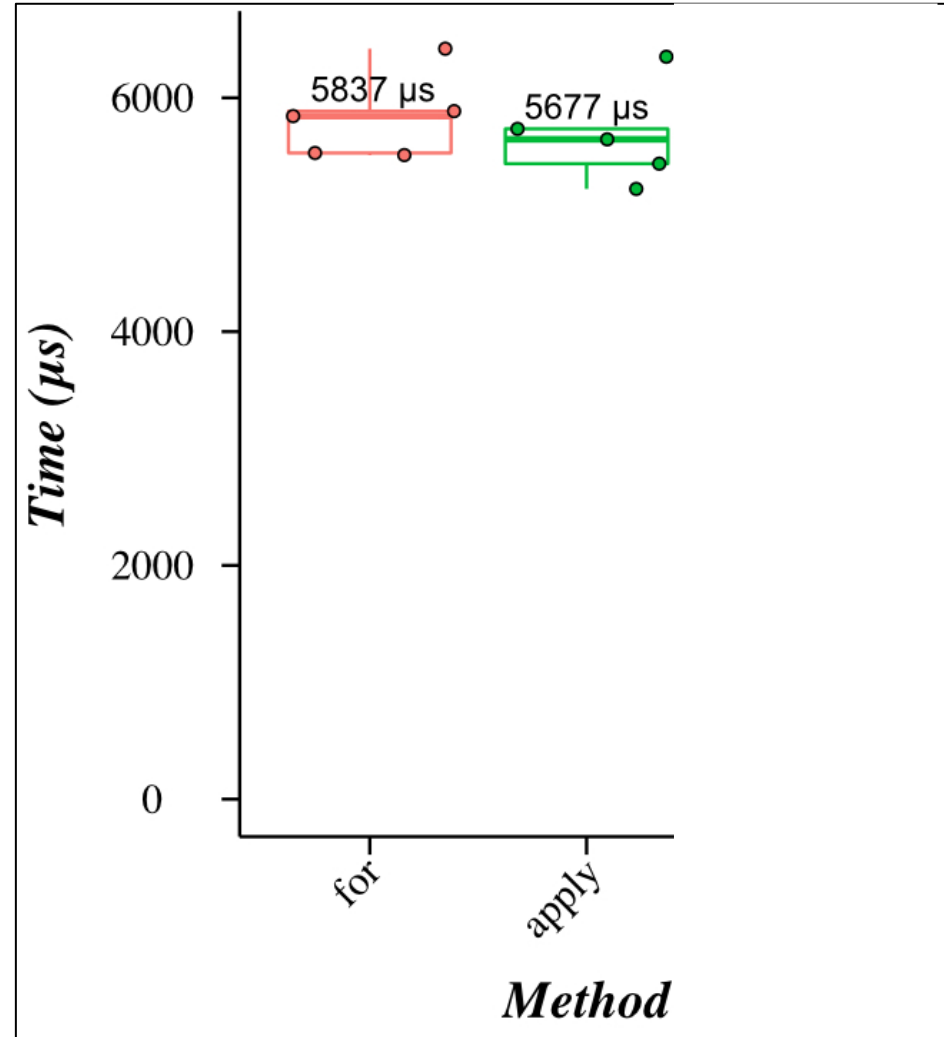
for() vs apply()

```
# chr1:1200312511-1587831227" return -387518716
```

```
# for loop
res_for <- rep(NA,length(genomePosition))
for(i in seq_along(genomePosition)){
  genomePos <- base::substring(text =
genomePosition[i],first = base::gregexpr(pattern =
":", text = genomePosition[i], fixed = T)[[1]][1]+1)
  minus <- base::gregexpr(pattern = "-",text =
genomePos, fixed = T)[[1]][1]
  res_for[i] <- as.integer(substring(text =
genomePos,first = 1,last = minus-1))-
as.integer(substring(text = genomePos,first =
minus+1))
}
```

```
# apply
res_apply <- sapply(X = genomePosition,FUN =
find_length,USE.NAMES = F)
```

```
# smart : define vectorized function that iterates
by itself i.e. no need for apply or for
res_smart <- find_length_base(genomePos =
genomePosition)
```



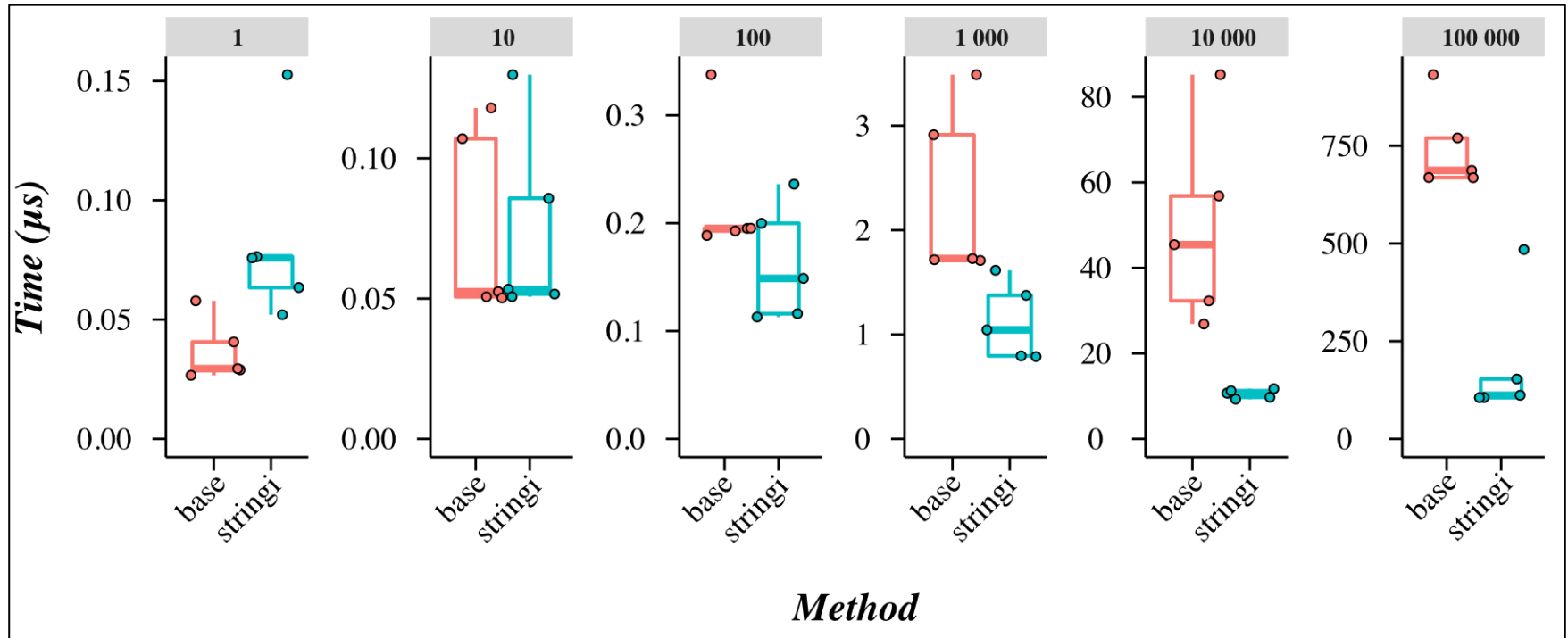
Even Faster string manipulation

```
# chr1:1200312511-1587831227" return -387518716
```

```
# base
find_length_base <- function(genomePos = NULL){
  genomePos <- base::substring(text = genomePos, first =
{unlist(base::gregexpr(pattern = ":", text = genomePos, fixed = T), use.names = F)+1})
  minus <- unlist(base::gregexpr(pattern = "-", text = genomePos, fixed = T), use.names
= F)
  return(as.integer(base::substring(text = genomePos, first = 1, last = minus-1))-
as.integer(base::substring(text = genomePos, first = minus+1)))
}
```

```
# stringi
find_length_stringi <- function(genomePos = NULL){
  genomePos <- stringi::stri_sub(str = genomePos, from
= stringi::stri_locate_first_fixed(str = genomePos, pattern = ":")[, 'start']+1, to = -
1)
  minus <- stringi::stri_locate_first_fixed(str = genomePos, pattern = "-")[, 'start']
  return(as.integer(stringi::stri_sub(str = genomePos, from=1, to = minus-1))-
as.integer(stringi::stri_sub(str = genomePos, from=minus+1, to = -1)))
}
```

String manipulation base vs stringi



for vs foreach

For a given mutation "GENE A123R" → "lung:7387, kidney:4, prostate:4, breast:2, oesophagus:2, small intestine:2, adrenal gland:1"

foreach

```
myCluster <- makeCluster(4, type = "FORK",useXDR=F,.combine=cbind)
registerDoParallel(myCluster)
results_1 <- foreach(mut = mutations[1:i],.combine = cbind,.inorder = T) %dopar% {
  var1 <- plyr::count(dat[dat$mutID==mut,],"Primary.site")
  var1 <- var1[order(var1$freq,decreasing = T),]
  return(list(mut, sum(var1$freq), stringi::stri_paste(var1$Primary.site, ":", var1$freq,
collapse=', ')))
}
stopCluster(myCluster)
```

foreach.fast : use the .inorder = F if you do not care about the order

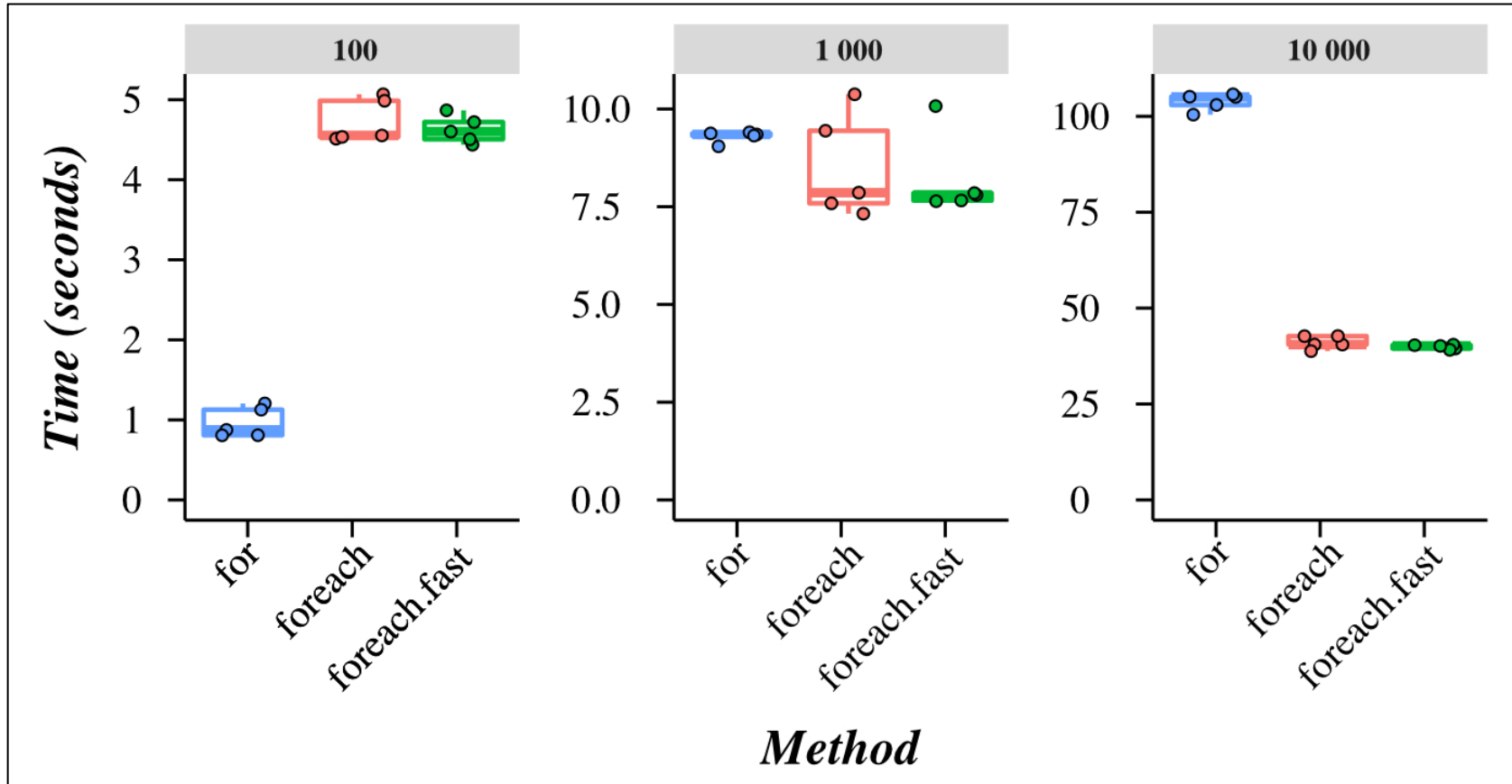
... same as "foreach" block ...

```
results_2 <- foreach(mut = mutations[1:i],.combine = cbind,.inorder = F) %dopar% {
  # ... same as "foreach" block ...
}
```

for

```
results_3 <- list()
for(mut in mutations[1:i]){
  var1 <- plyr::count(dat[dat$mutID==mut,],"Primary.site")
  var1 <- var1[order(var1$freq,decreasing = T),]
  results_3 <- cbind(results_3,list(mut, sum(var1$freq), stringi::stri_paste(var1$Primary.site,
":", var1$freq,collapse=', ')))
}
```

for vs foreach



lapply vs parLapply vs mclapply

Calculate mean FATHMM score for each gene

lapply

```
res_l <- base::lapply(X = unique(dat$Gene.name), FUN = function(X)  
  mean(dat[dat$Gene.name == X, "FATHMM.score"], na.rm = T))
```

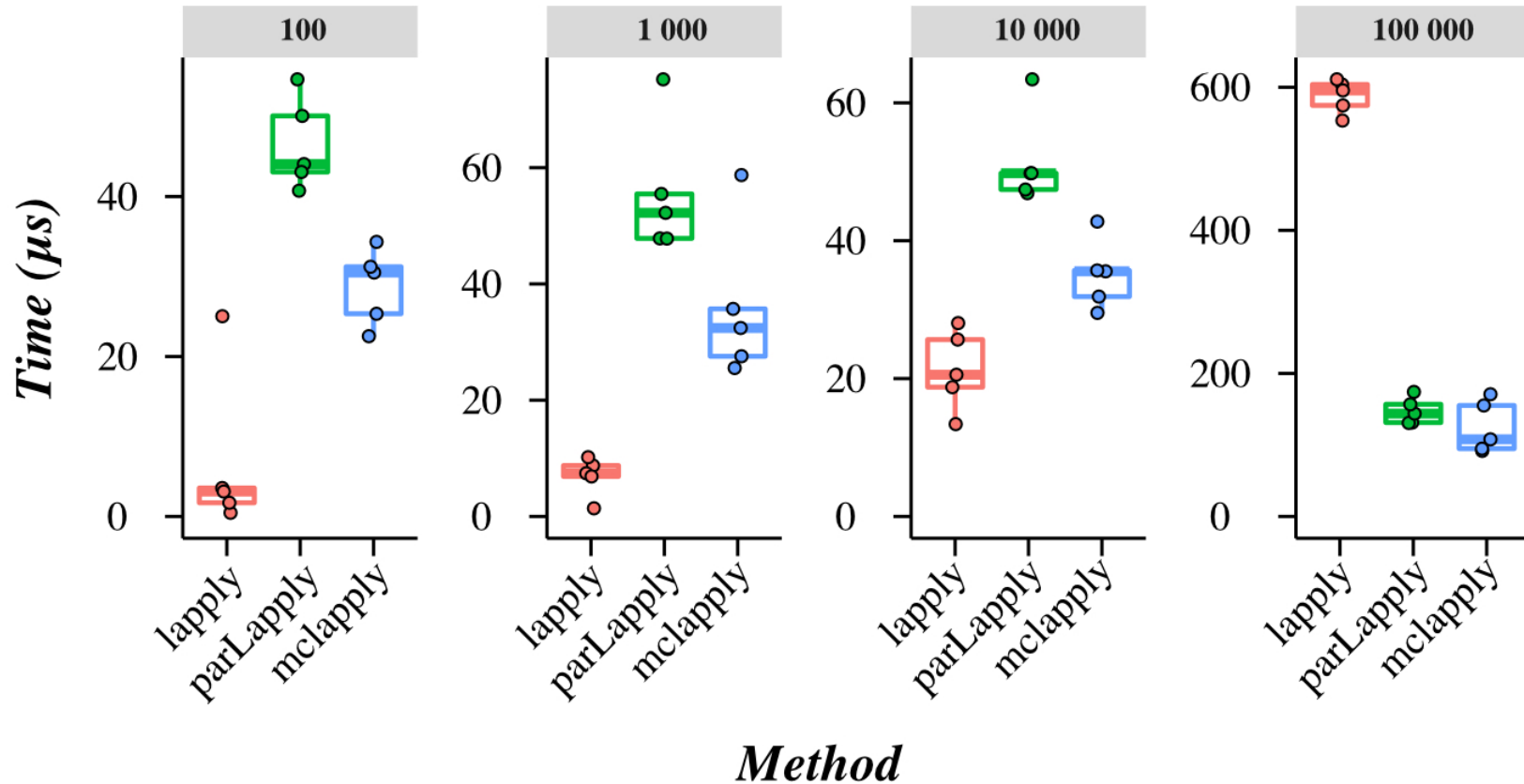
parLapply (snow)

```
myCluster <- makeCluster(4, type = "FORK", useXDR=F, .combine=cbind)  
registerDoParallel(myCluster)  
res_par <- snow::parLapply(cl = myCluster, X = unique(dat$Gene.name),  
  fun = function(X) mean(dat[dat$Gene.name == X, "FATHMM.score"], na.rm = T))  
stopCluster(myCluster)
```

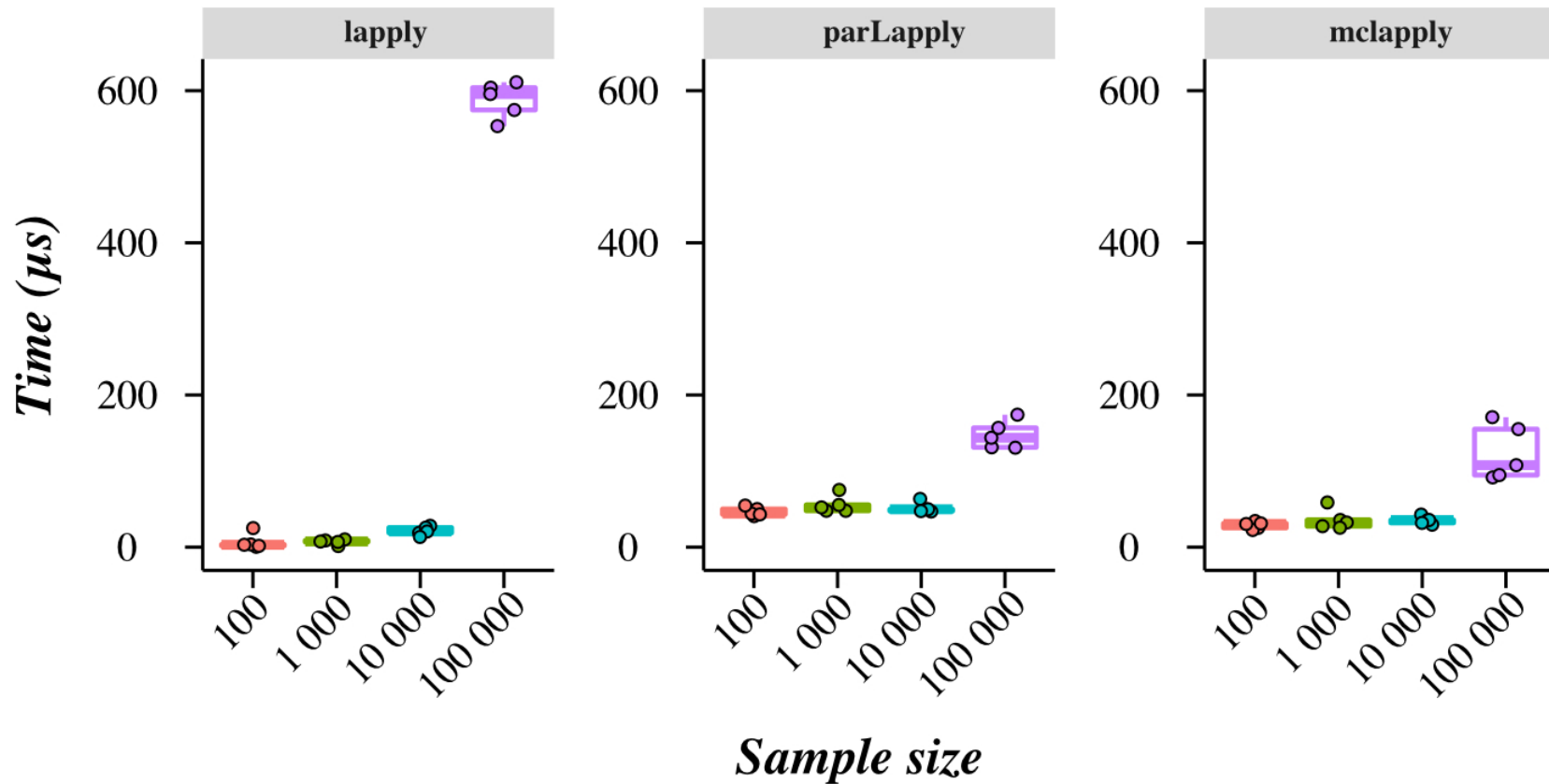
mclapply (parallel)

```
res_mcl <- parallel::mclapply(X = unique(dat$Gene.name),  
  FUN = function(X) mean(dat[dat$Gene.name==X, "FATHMM.score"], na.rm = T),  
  mc.cores = parallel::detectCores())
```

lapply vs parLapply vs mclapply



Go parallel when there is “*critical mass*”



saveRDS using multiple cores

Parallelized version of saveRDS (uses parallelized .gz compression)

Get it: <https://gist.githubusercontent.com/dchakro/8b1e97ba6853563dd0bb5b7be2317692/raw/parallelRDS.R>

Bottleneck: disk I/O

```
# base::saveRDS
```

```
base::saveRDS(object = dat,  
              compress = "gzip",  
              file = "/dev/null")
```

```
# parallelized saveRDS
```

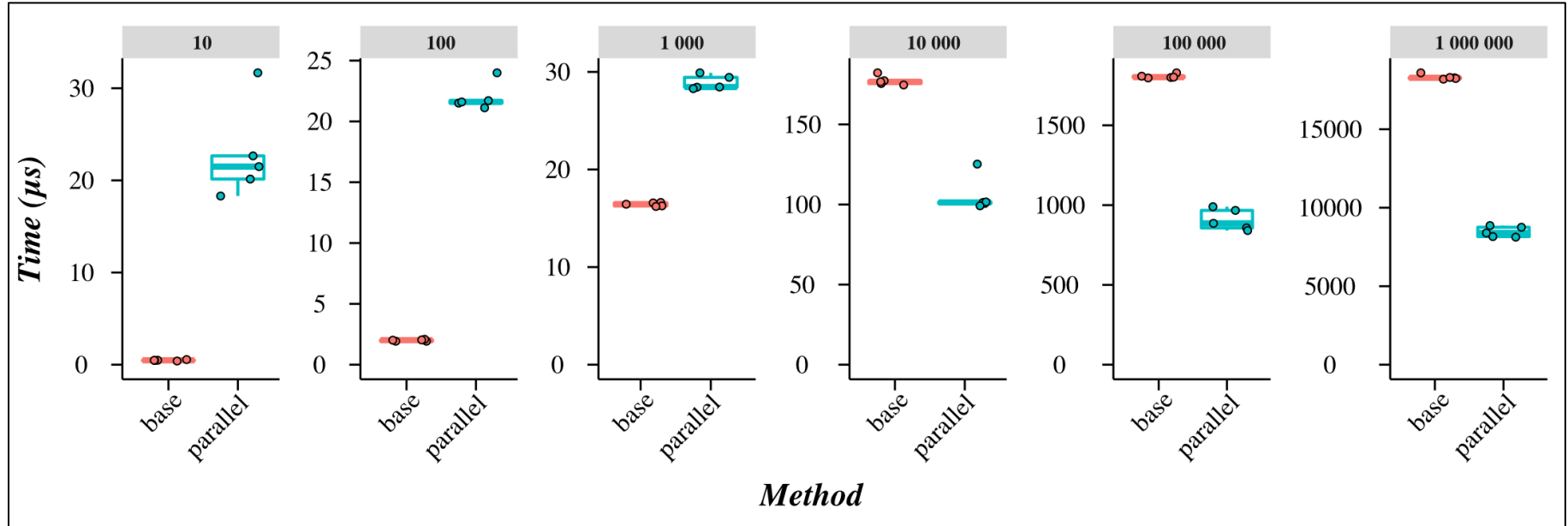
```
source("https://gist.githubusercontent.com/dchakro/8b1e97ba6853563dd0bb5b7be2317692/  
raw/parallelRDS.R")
```

```
saveRDS.gz(object = dat,  
           threads = parallel::detectCores(),  
           compression_level = 6,  
           file = "/dev/null")
```

saveRDS using multiple cores

Parallelized version of saveRDS (uses parallelized .gz compression)

Get it: <https://gist.githubusercontent.com/dchakro/8b1e97ba6853563dd0bb5b7be2317692/raw/parallelRDS.R>



Note:

- It is possible to use `-T` in `xz` for multi-threaded compression (since xz version 5.2)
 - But the decompression (reading) isn't multi-threaded.
- <https://github.com/vasi/pixz> - offers parallelized decompression.

Did you know ?

`utils::install.packages()` & `utils::update.packages` support multiple processors in R.

Define the number of cores you want to use with the `Ncpus` parameter.

```
install.packages(c("affy","stringi","ggplot2","data.table"),  
  ...other arguments ... ,  
  Ncpus = parallel::detectCores())
```

```
update.packages(...package list... ,  
  ...other arguments ... ,  
  Ncpus = parallel::detectCores())
```


{ } vs () – debunked ?

```
# parenthesis
pM <- rep(NA, length(dat$Gene.name))
for(i in seq_along(dat$Gene.name)){

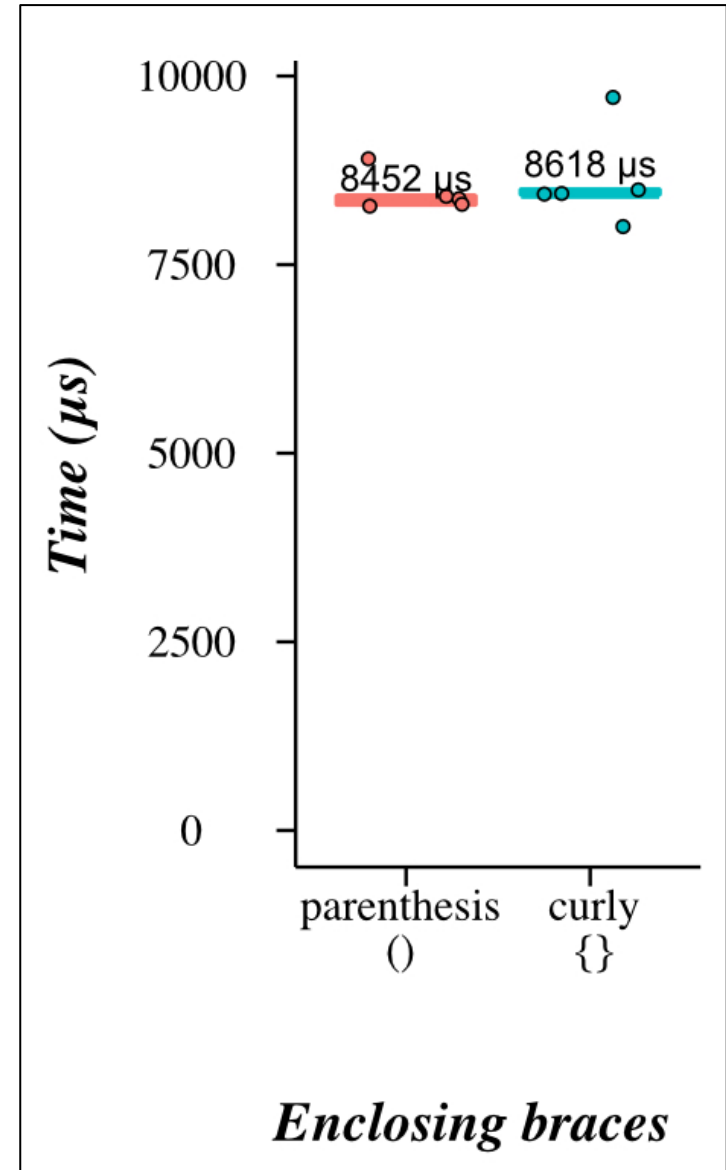
  pM[i] <- ((dat$ID_tumour[i] + dat$ID_sample[i] +
dat$MUTATION_ID[i]) / 3)

}

# curly
cM <- rep(NA, length(dat$Gene.name))
for(i in seq_along(dat$Gene.name)){

  cM[i] <- {{dat$ID_tumour[i] + dat$ID_sample[i] +
dat$MUTATION_ID[i]}} / 3

}
```

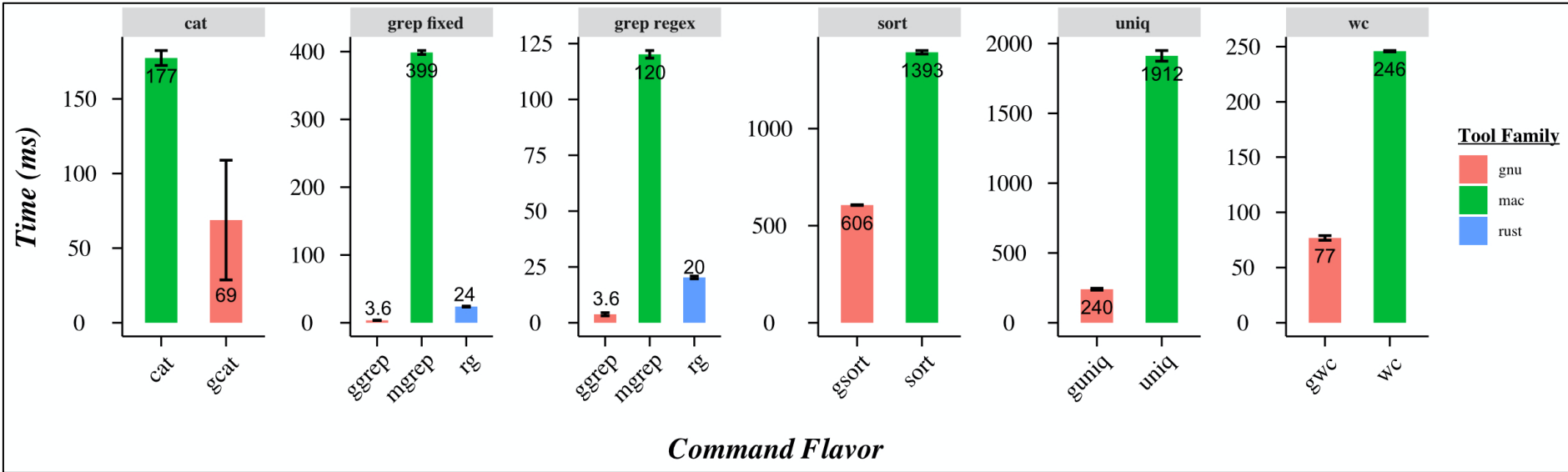


Summary / Tips *(may depend on your use case)*

- Use vectorization
 - Use apply family of functions instead of iterating
- Remember, to check if you benefit from parallel computing
- Write a `function()` when possible:
 - Functions are compiled into byte-code by the JIT compiler in R
 - JIT is enabled by default since R version 3.4
- JIT also compiles `for()`, `while()` and `repeat()` structures
- Check function documentation for tips (e.g. `unlist` + `use.names`)
- Try to compile packages from source (optimized for your system).

Bonus tips *(for Mac users)*

- Install Homebrew (<https://brew.sh>), and then install:
 - `brew install coreutils grep ripgrep`
- To log brew activity you can use: github.com/dchakro/brewlog
 - `brewlog install coreutils grep ripgrep`



Thank you!

Additional resources

- Checkout R packages: `profvis`, `microbenchmark`, `bench::mark()`
- <http://www.dartistics.com/fast-r-code.html>
- <https://csgillespie.github.io/efficientR/performance.html>
- <http://adv-r.had.co.nz/Performance.html>
- <https://www.r-bloggers.com/strategies-to-speedup-r-code/>
- <https://www.r-bloggers.com/faster-higher-stronger-a-guide-to-speeding-up-r-code-for-busy-people/>
- <https://cran.r-project.org/web/packages/Rfast/index.html>