Mopsgeschwindigkeit

Der Code in `slow-sim.R` implementiert eine (relativ sinnbefreite) Simulationsstudie um die Verteilung der geschätzten Regressionskoeffizienten $\hat{\beta}$ in einem Modell $y \sim t(\text{ncp} = X\beta, \text{df} = 4)$ mit $t(4)$-verteilten Fehlern und linearem Prädiktor $X\beta$ zu bestimmen:

```
source("slow-sim.R")

set.seed <- 232323
observations <- 5000
covariates <- 10
testdata <- as.data.frame(
  matrix(rnorm(observations * covariates),
         nrow = observations
  ))

test <- simulate(reps = 100, seed = 20141028, data = testdata)

system.time(test <- simulate(reps = 100, seed = 20141028, data = testdata))
```

```
##    user  system elapsed
##   1.735   3.238   0.826
```

Die Simulation ist recht ineffizient programmiert.

a) Benutzen Sie die in der Vorlesung kennengelernten Profiling-Methoden um die Stellen zu identifizieren an denen das Skript in `slow-sim.R` die meiste Zeit verbringt.
b) Modifizieren Sie den Code in `slow-sim.R` so, dass er i) **mindestens 5x schneller** läuft (ohne dass sich die Ergebnisse qualitativ ändern!!) und ii) unseren Vorstellungen von sauber dokumentierter, gut strukturierter und defensiv programmierter Software entspricht.

Hinweis: Betrachten Sie zu a) sowohl wo in dem Code von `slow-sim.R` die meiste Zeit verbraucht wird als auch welche *eingebauten* R-Funktionen dort aufgerufen werden und was diese tun und wie.
Für b) sollten Sie sich zuerst mal überlegen was man hier eigentlich tun will ("First, solve the problem. Then, write the code.") um dann kritisch auf den Code zu gucken: Wo tut er mehr als er eigentlich muss? Wo wiederholt sich Schritte überflüssigerweise? Können Sie Berechnungen vektorisieren oder Zuweisungen prä-allozieren?
Wenn Sie den Code in b) schön effizient gemacht haben versuchen Sie auch noch ihn (möglichst: plattformunabhängig) zu parallelisieren, mit einem Paket Ihrer Wahl. (Obacht: `future` funktioniert nicht unbedingt verläßlich in RStudio, benutzen Sie da zum Testen eine normale R-Konsole. . . .)

Lösung:

Hinweis: Die Ergebnisse des Profilings werden auf Ihren Rechnern nur so ähnlich aussehen wie die unten abgebildeten, nicht genau gleich – profiiling ist erstens ein stochastisches Verfahren, und zweitens beeinflussen viele Aspekte Ihres Systems die Rechenzeiten. . . .

a)

Wir benutzen also den visuellen Profiler (mittels `profvis` oder auch Menüpunkt "Profile" in RStudio):

```
library(profvis)
profvis(test <- simulate(reps = 100, seed = 20141028, data = testdata))
```

Aha – wir sehen also in Fig. 1 dass der Code die komplette Zeit in `cbind` verbringt, was daran liegt dass die Zeile `coefs <- cbind(coefs, simulate_once(data, true_coef, df))` quasi alle weiteren von uns definierten/benutzten Funktionen aufruft die nennenswert Rechenzeit benötigen. **Hier könnten wir evtl. durch *pre-allocation* von `coefs` ein bisschen Zeit sparen und Duplikationen vermeiden.**

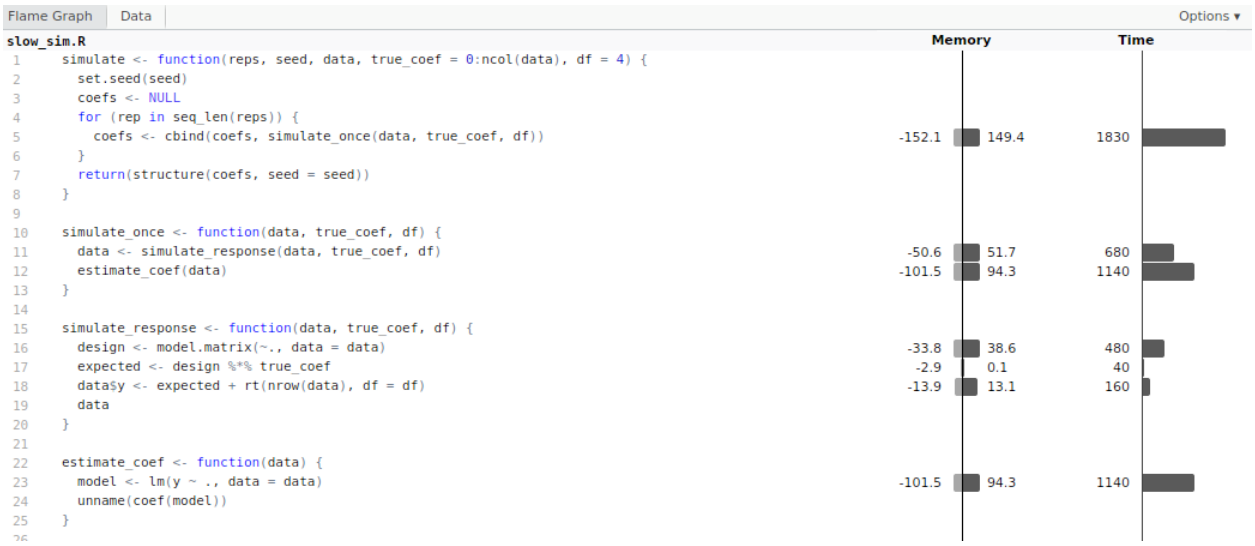

Figure 1: Flame-Graph für `profvis(test <- simulate(reps = 100, seed = 20141028, data = testdata))`

Wenn wir uns das profiling für `simulate_once`, also die Funktion die eigentlich die ganze Arbeit macht, ansehen, lernen wir, dass unser Simulationscode etwas mehr als ein Drittel der Zeit damit verbringt neue Responsevektoren zu erzeugen (Zeile 11, `simulate_response`-Aufruf) und den Rest damit auf das Modell für diese zu schätzen (Zeile 12, `estimate_coef`-Aufruf).

Wenn man sich anschaut wie sich der Aufwand für die Erzeugung der Responsevektoren verteilt ist (also das Profiling innerhalb von `simulate_response`) sieht man dass über ein Viertel der Gesamt-Zeit dafür verbraucht wird immer und immmer wieder die selbe Designmatrix X (Zeile 16) und den selben Vektor $E(y)$ (Zeile 17) zu erzeugen , `data` und `true_coef` bleiben ja schließlich in jeder Replikation gleich und nur y wird neu erzeugt. **Das sollten wir also einfach einmal berechnen und dann in jeder Replikation wieder verwenden um den Code schneller zu machen.**
Etwa ein Zehntel der Gesamtzeit geht damit drauf in jeder Replikation t-verteilte Fehler zu erzeugen (Zeile 18), da können wir wohl nicht viel machen und selbst wenn würde sich hier der Aufwand kaum lohnen.

Wenig überraschend ist, dass wir mit Abstand die meiste Zeit dafür verbrauchen in jeder Replikation das lineare Modell zu schätzen. Hier lohnt es sich also mal genauer nachzusehen was dort in Zeile 23 so lange dauert. Dafür können wir entweder in dem "Data"-Tab des Profile-Fensters den *call stack* für den `lm`-Aufruf inspizieren indem wir auf die entsprechenden Dreiecke klicken (s. Fig 2).

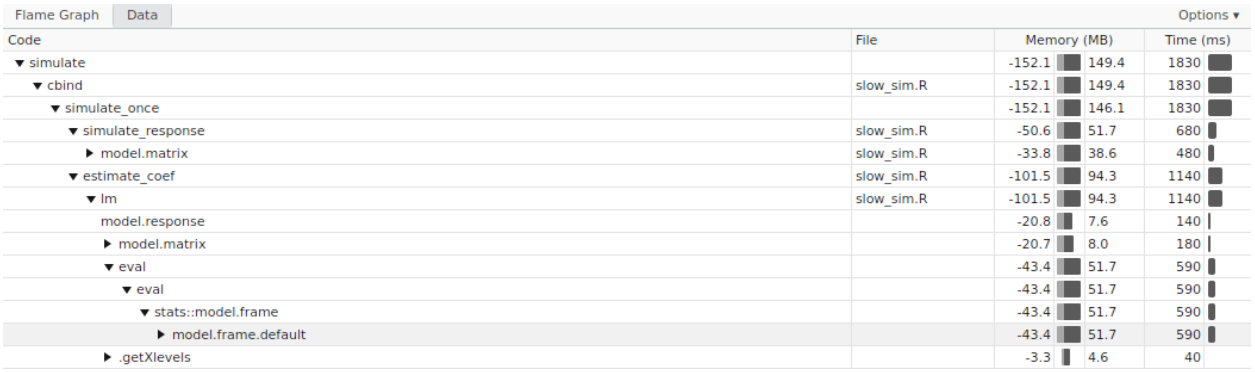

Figure 2: Data-Tab für `profvis(test <- simulate(reps = 100, seed = 20141028, data = testdata))`

Oder im Flame-Graph unten in der Zeitachse auf einen der `lm`-Aufrufe doppelklicken um dort in die Zeitachse hinein zuzoomen (s. Fig. 3).

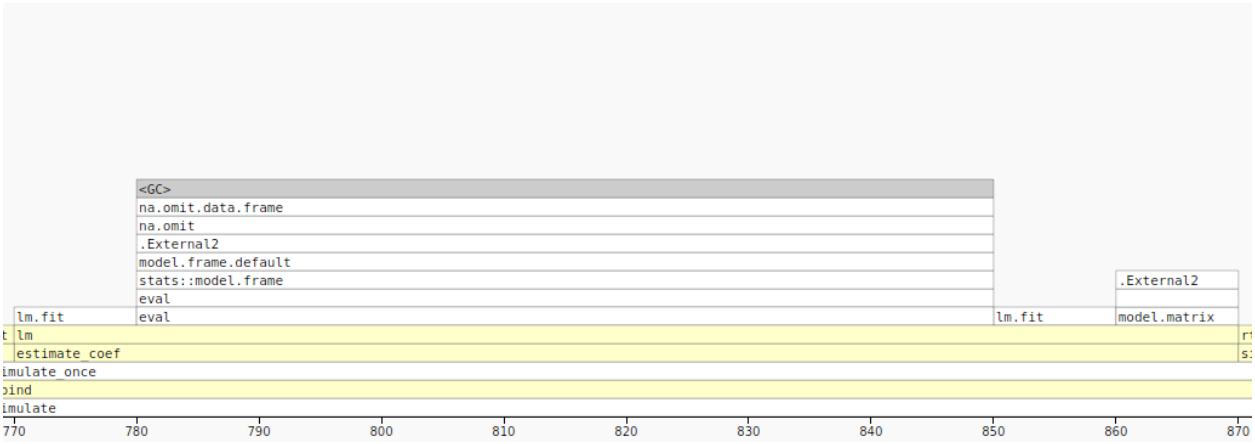

Figure 3: Zoom in die Zeitachse des Flame-Graph für `profvis(test <- simulate(reps = 100, seed = 20141028, data = testdata))`

(An der Stelle könnte es helfen sich auch mal den Code für die `lm`-Funktion durchzulesen und die Hilfe der Funktionen im *call stack* die sie nicht kennen zu überfliegen.) In beiden Fällen erkennen wir dass die Funktion `lm` einen Großteil ihrer Zeit mit Aufrufen von `model.frame` beschäftigt ist. Diese Hilfsfunktion entfernt `NA`s aus dem Datensatz, prüft ob alle Variablen in der Formel auch wirklich im Datensatz oder der Formelumgebung vorhanden sind, etc… – alles Dinge die wir hier nicht wirklich brauchen! Wir können stattdessen direkt `lm.fit()` aufrufen um das Modell zu schätzen und auf diese input checks und Vorverarbeitungsschritte verzichten da wir ja eh nur einen sauberen, selbst generierten Datensatz von bester Qualität an `lm` übergeben. `lm.fit()` bekommt als Inputs statt Daten und einer Modellformel direkt die Designmatrix und den Responsevektor, und ersteres wollten wir ja sowieso einmal im Voraus berechnen um Zeit bei der Erzeugung der Responsevektoren zu sparen. Die Hilfe von `lm` sagt dazu: "`lm` calls the lower level functions `lm.fit`[…] for the actual numerical computations. For programming only, you may consider doing likewise."

b)

In `faster-sim.R` findet sich Code (hier aufgeführt) der die oben festgestellten Flaschenhäse & Ineffizienzen beseitigt und ordentliche *input checks* und Dokumentation hat:

```r
# simulate and estimate linear model with t-distributed errors for a fixed design
# inputs: reps: how many replications?
#         seed: RNG seed
#         data: data.frame containing all and only numeric covariates
#         true_coefs: coefficient vector to use for simulating new responses
#         df: degrees of freedom of the residual error t-distribution
# output: a matrix of coefficient vectors (each column is one replicate), with
#         attribute "seed" = RNG seed of the generating call for reproducibility.
simulate_faster <- function(reps, seed, data, true_coef = 0:ncol(data), df = 4L) {
  # change 0: input checks
  check_simulate_inputs(reps, seed, data, true_coef, df)

  set.seed(seed)
  # change 1: pre-allocate container for results
  coefs <- matrix(0, nrow = length(true_coef), ncol = reps)

  # change 2a: pre-compute design X, expected (X*beta):
  design <- model.matrix(~., data = data)
```

```
  expected <- design %*% true_coef

  for (rep in seq_len(reps)) {
    coefs[, rep] <- simulate_once_faster(expected, design, df)
  }
  return(structure(coefs, seed = seed))
}

simulate_once_faster <- function(expected, design, df) {
  response <- simulate_response_faster(expected, df)
  estimate_coef_faster(response, design)
}

simulate_response_faster <- function(expected, df) {
  # change 2b: reuse <expected>, don't re-compute
  expected + rt(length(expected), df = df)
}

estimate_coef_faster <- function(response, design) {
  # change 3: .lm.fit instead of lm to avoid unnecessary input checks
  model <- .lm.fit(y = response, x = design)
  coef(model)
}

# change 0: input checks
check_simulate_inputs <- function(reps, seed, data, true_coef, df) {
  checkmate::assert_count(reps)
  checkmate::assert_integer(seed, lower = 0)
  checkmate::assert_data_frame(data, types = "numeric")
  checkmate::assert_numeric(true_coef,
    finite = TRUE, any.missing = FALSE,
    len = ncol(data) + 1
  )
  checkmate::assert_numeric(df, lower = 0)
}
```

```
source("faster-sim.R")
all.equal(
  simulate(reps = 10, seed = 20141028L, data = testdata),
  simulate_faster(reps = 10, seed = 20141028L, data = testdata)
)
```

```
## [1] TRUE
```

```
bench::mark(
  slow = simulate(reps = 100, seed = 20141028L, data = testdata),
  faster = simulate_faster(reps = 100, seed = 20141028L, data = testdata),
  min_iterations = 10
)
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is disabled.
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
```

```
## 1 slow          836ms    1.04s      0.915   521.5MB    13.6
## 2 faster        180ms 188.29ms      5.10     55.9MB     7.64
```

Das hat sich also durchaus gelohnt – fast 10x weniger Speicherverbrauch (!) und gleichzeitig auch noch deutlich schneller.

Schauen wir uns an wie jetzt die Rechenzeit verteilt ist:

```
profvis::profvis(
  test_faster <- simulate_faster(reps = 100, seed = 20141028L, data = testdata))
```

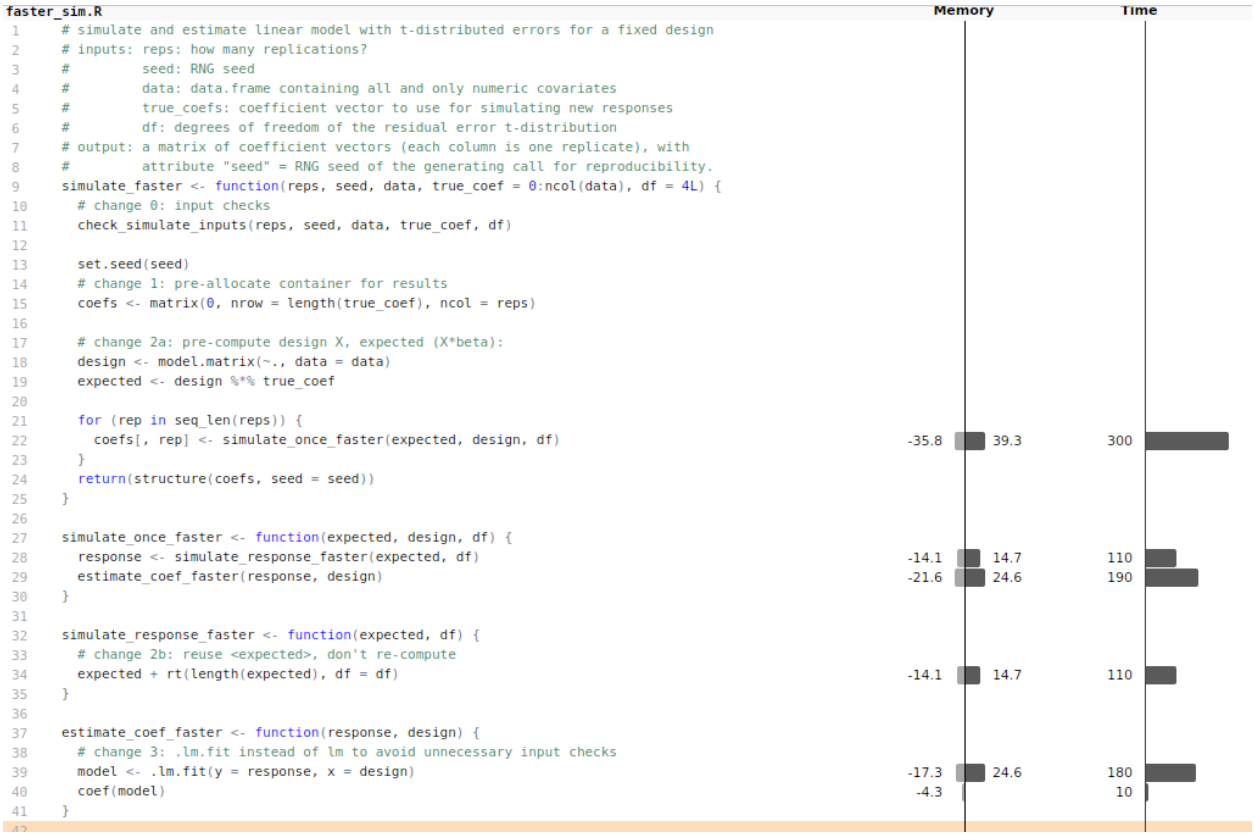

Figure 4: Flame-Graph für `profvis(test_faster <- simulate_faster(reps = 100, seed = 20141028L, data = testdata))`

Das macht jetzt mehr Sinn als vorher, s. Fig. 4: Wir verbringen etwa zwei Drittel der Zeit damit das Modell zu schätzen (Zeile 29) und den Rest der Zeit damit t-verteilte Zufallsvariablen zu generieren (Zeile 28 bzw. 34). Alle anderen Operationen brauchen (fast) keine messbare Rechenzeit mehr.
Beachten Sie auch wieviel speichereffizienter jetzt die Zuweisung an `coefs` geworden ist: Wenn wir Koeffizientenvektoren wie hier in eine bestehende Matrix der richtigen Größe schreiben (`coefs[, rep] <- simulate_once_faster(expected, design, df)`) statt in jeder Iteration eine neue Spalte an eine Matrix anzuhängen (`coefs <- cbind(coefs, simulate_once(data, true_coef, df))`) brauchen wir viel weniger Schreib- und Löschoperationen im Speicher (Memory-Spalte hier bei Zeile 22 verglichen mit Zeile 5 in Figure 1)...

Mehr Möglichkeiten für bessere Performance:

Möglichkeit 1: Parallelisierung

Wie fast alle Simulationsstudien ist das hier natürlich *embarrassingly parallel*, das heisst so etwas wie der

Code weiter unten wäre eine Möglichkeit weitere Zeit zu sparen wenn man mehrere Prozesse gleichzeitig laufen lassen kann:

```
# simulate and estimate linear model with t-distributed errors for a fixed design
# inputs: reps: how many replications?
#         seed: RNG seed
#         data: data.frame containing all and only numeric covariates
#         true_coefs: coefficient vector to use for simulating new responses
#         df: degrees of freedom of the residual error t-distribution
#         cluster: (optional) a SOCKET Cluster for parallel::parLapply
#         cores: (optional) the number of processes to use for parallelization
# output: a matrix of coefficient vectors (each column is one replicate), with
#         attribute "seed" = RNG seed of the generating call for reproducibility.
simulate_parallel <- function(reps, seed, data, true_coef = 0:ncol(data),
                              df = 4, cluster = NULL, cores = NULL) {
  library(parallel)
  check_simulate_inputs(reps, seed, data, true_coef, df)
  set.seed(seed)
  # make sure RNG is set to parallel mode:
  RNGkind("L'Ecuyer-CMRG")
  if (is.null(cores)) {
    # use one less process than maximum so computer is not completely busy
    cores <- detectCores() - 1L
  }
  checkmate::assert_integerish(cores, lower = 1)

  design <- model.matrix(~., data = data)
  expected <- design %*% true_coef

  # parallel apply-function is platform-dependent, compare boot::boot()
  # for similar code...
  if (.Platform$OS.type != "windows") {
    # simply use forking (i.e.: mclapply) for parallelization for unix, mac:
    papply <- function(X, FUN, ...) {
      mclapply(X = X, FUN = FUN, mc.cores = cores, mc.set.seed = TRUE, ...)
    }
  } else {
    # use socket clusters on windows
    if (is.null(cluster)) {
      cluster <- makePSOCKcluster(cores)
      # ... and do clean up after yourself:
      on.exit(stopCluster(cl = cluster))
    }
    checkmate::assert_class(cluster, "cluster")
    # make sure RNG is set to parallel mode:
    clusterSetRNGStream(cluster)
    # define parallel-apply function for windoze:
    papply <- function(X, FUN, ...) {
      parLapply(cl = cluster, X = X, fun = FUN, ...)
    }
    # load needed objects onto cluster:
    clusterExport(
      cl = cluster,
      varlist = c(
```

```
        "simulate_once_faster", "simulate_response_faster",
        "estimate_coef_faster", "expected", "design", "df"
      ),
      # variables are found in the execution environment of the call of this
      # function and in its parents. The code below returns the environment that
      # the call is being evaluated in at runtime, see ?sys.nframe
      # the terrible(!) default for clusterExport is to use ".GlobalEnv"
      # to look for the variables in varlist, so ALWAYS set this explicitly:
      envir = sys.frame(sys.nframe())
    )
  }
  # perform parallelization by using parallel-apply on vector [1, 2, ...., reps]
  #  calling the simulate_once function (with identical arguments) each time.
  coefs <- papply(
    X = seq_len(reps),
    FUN = function(i, expected, design, df) {
      unname(simulate_once_faster(expected, design, df))
    },
    expected = expected, design = design, df = df
  )
  coefs <- do.call(cbind, coefs)
  return(structure(coefs, seed = seed))
}

# das Selbe in Grün mit foreach:
simulate_foreach <- function(reps, seed, data, true_coef = 0:ncol(data),
                             df = 4, cluster = NULL, cores = NULL) {
  library(foreach)
  library(doParallel)
  library(doRNG) #
  check_simulate_inputs(reps, seed, data, true_coef, df)
  set.seed(seed)
  if (is.null(cores)) {
    # use one less process than maximum so computer is not completely busy
    cores <- detectCores() - 1L
  }
  checkmate::assert_integerish(cores, lower = 1)

  design <- model.matrix(~., data = data)
  expected <- design %*% true_coef

  # initialize parallel backend:
  registerDoParallel(cores = cores)
  coefs <- matrix(0, nrow = length(true_coef), ncol = reps)
  coefs <- foreach(rep = seq_len(reps),
    .export = c("simulate_once_faster", "simulate_response_faster",
        "estimate_coef_faster", "expected", "design", "df"),
    #.export only necessary under windows....
    .combine = cbind) %dorng% simulate_once_faster(expected, design, df)
  # delete parallel seeds:
  attr(coefs, "rng") <- NULL
  return(structure(unname(coefs), seed = seed))
}
```

```r
# das Selbe in Grün mit future:
simulate_future <- function(reps, seed, data, true_coef = 0:ncol(data),
                            df = 4, cluster = NULL, cores = NULL) {

  check_simulate_inputs(reps, seed, data, true_coef, df)
  set.seed(seed)
  if (is.null(cores)) {
    # use one less process than maximum so computer is not completely busy
    cores <- detectCores() - 1L
  }
  checkmate::assert_integerish(cores, lower = 1)

  design <- model.matrix(~., data = data)
  expected <- design %*% true_coef

  # initialize parallel backend:
  future::plan("multiprocess", workers = cores)

  coefs <- future.apply::future_replicate(n = reps,
    expr = simulate_once_faster(expected, design, df))

  return(structure(unname(coefs), seed = seed))
}

# mclapply-ergebnisse sind nicht wirklich reproduzierbar
# (sollte unter windows mit socket-cluster aber funktinonieren...)
all.equal(
  simulate_parallel(reps = 10, seed = 20141028L, data = testdata),
  simulate_parallel(reps = 10, seed = 20141028L, data = testdata)
)
```

```
## [1] "Mean relative difference: 0.004468819"
```

```r
# foreach mit doRNG ist reproduzierbar:
all.equal(
  simulate_foreach(reps = 10, seed = 20141028L, data = testdata),
  simulate_foreach(reps = 10, seed = 20141028L, data = testdata)
)
```

```
## Loading required package: iterators
```

```
## Loading required package: rngtools
```

```
## [1] TRUE
```

```r
# future ist verläßlich reproduzierbar:
all.equal(
  simulate_future(reps = 10, seed = 20141028L, data = testdata),
  simulate_future(reps = 10, seed = 20141028L, data = testdata)
)
```

```
## [1] TRUE
```

```r
# bench::mark does not work (well) for parallelized stuff....
rbenchmark::benchmark(
  faster   = simulate_faster(reps = 100, seed = 20141028L, data = testdata),
  parallel = simulate_parallel(reps = 100, seed = 20141028L, data = testdata, cores = 4),
```

```
  foreach  = simulate_foreach(reps = 100, seed = 20141028L, data = testdata, cores = 4),
  future   = simulate_future(reps = 100, seed = 20141028L, data = testdata, cores = 4),
  replications = 10, columns = c("test", "elapsed", "relative")
)
```

```
##       test elapsed relative
## 1   faster   2.439    2.484
## 3  foreach   1.479    1.506
## 4   future   1.603    1.632
## 2 parallel   0.982    1.000
```

Obacht: Parallelisierte Zufallsgeneratoren generieren andere Zufallszahlen!

Ergebnis hier unter Benutzung von 4 Prozessen.
Der Komfort von `foreach` oder `future` kostet halt auch ein bißchen Performance...

Möglichkeit 2: Spezialisierte Pakete

`RcppArmadillo` hat eine Funktion `fastLm()` bzw. `fastLmPure()`, mal schauen was die kann:

```
estimate_coef_rcpp <- function(response, design) {
  model <- RcppArmadillo::fastLmPure(X = design, y = response)
  model$coefficients[, 1] #cast to vector
}

design <- model.matrix(~., data = testdata)
response <- rnorm(nrow(testdata))
bench::mark(
  faster = estimate_coef_faster(response, design),
  rcpp = estimate_coef_rcpp(response, design),
  min_iterations =  10
)
```

```
## # A tibble: 2 x 6
##   expression      min   median ‘itr/sec‘ mem_alloc ‘gc/sec‘
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 faster        530us  705.5us     1317.  508.18KB     12.1
## 2 rcpp          965us   1.04ms      414.    4.36MB      0
```

OK – modernste C++-Magie hier nicht zielführend, für größere Daten / komplexere Modelle kann das durchaus was bringen. Außerdem hier gut sichtbar: das “memory profiling” von `{bench}` ist irreführend bei Code der externe C++-Bibliotheken aufruft, `{RcppArmadillo}` braucht natürlich nicht wirklich 0B Speicher, der wird aber dort verbraucht wo `{bench}` es nicht mitbekommt....

Und natürlich *last, but not at all least*:

Möglichkeit 3: “*First, understand the problem. Then, write the code*”

Was wir hier eigentlich in jeder Replikation nur tun müssen um die Koeffizienten zu bekommen, ist die gute alte Normalengleichung

$$\hat{\beta} = (X'X)^{-1}X'y$$

zu lösen. Der Teil $(X'X)^{-1}X'$ bleibt von Iteration zu Iteration gleich, den können wir also **einmal** vorberechnen und dann in jeder Iteration wiederverwenden – damit müssen wir nicht mehr in jeder Iteration ein Gleichungssystem lösen, sondern machen nur noch eine einfache Matrix-Vektor-Multiplikation – also:

```
# Documentation: see simulate_faster()
simulate_even_faster <- function(reps, seed, data, true_coef = 0:ncol(data), df = 4L) {
```

```
  check_simulate_inputs(reps, seed, data, true_coef, df)

  set.seed(seed)
  coefs <- matrix(0, nrow = length(true_coef), ncol = reps)

  design <- model.matrix(~., data = data)
  expected <- design %*% true_coef
  # change: pre-compute factor (X'X)^-1 X' for normal equation
  #           beta_hat = (X'X)^-1 X'y
  # inverting X'X could go wrong, so capture errors via try
  coefficient_matrix <- try(solve(crossprod(design)) %*% t(design))
  if (inherits(coefficient_matrix, "try-error")) {
    stop("Could not solve normal equation. Design may not have full rank.")
  }

  for (rep in seq_len(reps)) {
    coefs[, rep] <- simulate_once_even_faster(expected, coefficient_matrix, df)
  }
  return(structure(coefs, seed = seed))
}

simulate_once_even_faster <- function(expected, coefficient_matrix, df) {
  response <- simulate_response_faster(expected, df)
  # change: get coefficients with simple matrix-vector-multiplication
  coefficient_matrix %*% response
}

all.equal(
  simulate_faster(reps = 10, seed = 20141028L, data = testdata),
  simulate_even_faster(reps = 10, seed = 20141028L, data = testdata)
)
```

```
## [1] TRUE
```

```
bench::mark(
  faster = simulate_faster(reps = 100, seed = 20141028L, data = testdata),
  even_faster =
    simulate_even_faster(reps = 100, seed = 20141028L, data = testdata),
  min_iterations = 50
)
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is disabled.
```

```
## # A tibble: 2 x 6
##   expression       min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>  <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 faster         190ms    206ms      4.57   55.62MB    5.67
## 2 even_faster    143ms    280ms      3.43    6.84MB    0.480
```

.. das bringt also nochmal nochmal bischn Geschwindigkeit, und vor allem auch weniger Speicherverbrauch.

Hinweis:

Wenn Sie mitgedacht haben (haben Sie etwa nicht...?!?), werden Sie sich an dieser Stelle fragen warum wir dann das nicht *noch* schneller machen indem wir, statt in einer Schleife `reps`-mal diese Matrix-Vektor-Multiplikation durchführen, einfach direkt ein einziges Mal eine Matrix, die in den Spalten `reps` zufällig generierte y-Vektoren enthält, an $(X'X)^{-1}X'$ dranmultiplizieren, also so etwas wie

```
errors <- matrix(rt(length(expected) * reps, df = df), ncol = reps)
responses_matrix <- matrix(expected, nrow = length(expected), ncol = reps) +
  errors
coefs <- coefficient_matrix %*% responses_matrix
```

Es stellt sich raus dass das tatsächlich nochmal schneller als die Variante oben wäre:

```
compute_looped <- function(coefficient_matrix, reps = 100) {
  n <- ncol(coefficient_matrix)
  for (r in seq_len(reps)) coefficient_matrix %*% rt(n, df = 4)
}
compute_once <- function(coefficient_matrix, reps = 100) {
  n <- ncol(coefficient_matrix)
  responses_mat <- matrix(rt(n * reps, df = 4), nrow = n)
  coefficient_matrix %*% responses_mat
}

n_obs <- 5e3
n_coefs <-  20
coefficient_matrix <- matrix(rnorm(n_obs * n_coefs), nrow = n_coefs)

bench::mark(
  loop = compute_looped(coefficient_matrix),
  once = compute_once(coefficient_matrix),
  min_iterations = 50, check = FALSE
)
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 loop          183ms    257ms      3.75    3.86MB    0.416
## 2 once          153ms    160ms      5.66    7.64MB    1.24
```

aber evtl für große Daten und große `reps` möglicherweise Probleme bereitet weil die erzeugte `nrow(data)xreps` Matrix `response_mat` mit den `response`-Vektoren eventuell zu groß wird um sie im Arbeitsspeicher zu halten.

Noch eine (theoretische) Möglichkeit: Byte-compilation

S. `library(compiler); ?cmpfun` bzw. das entsprechende Kapitel aus Colin Gillespie's and Robin Lovelace's "Efficient R Programming" hier. Bringt in diesem Fall wenig bis nichts, weil die relevanten Teile des Codes großteils sowieso schon optimierte, vor-kompilierte *high-level* Funktionen aus `base` und `stats` aufrufen.