Profiling & Parallelization

Lecture 21

Dr. Colin Rundel

Profiling & Benchmarking

profvis demo

```r
n = 1e6
d = tibble(
  x1 = rt(n, df = 3),
  x2 = rt(n, df = 3),
  x3 = rt(n, df = 3),
  x4 = rt(n, df = 3),
  x5 = rt(n, df = 3),
) |>
  mutate(y = -2*x1 - 1*x2 + 0*x3 + 1*x4 + 2*x5 + rnorm(n))
```

```r
profvis::profvis({
  lm(y~., data=d)
})
```

profvis demo 2

```r
profvis::profvis({
  data = data.frame(value = runif(5e4))

  data$sum[1] = data$value[1]
  for (i in seq(2, nrow(data))) {
    data$sum[i] = data$sum[i-1] + data$value[i]
  }
})
```

```r
profvis::profvis({
  x = runif(5e4)
  sum = x[1]
  for (i in seq(2, length(x))) {
    sum[i] = sum[i-1] + x[i]
  }
})
```

Benchmarking - bench

```
d = tibble(
  x = runif(10000),
  y = runif(10000)
)

(b = bench::mark(
  d[d$x > 0.5, ],
  d[which(d$x > 0.5), ],
  subset(d, x > 0.5),
  filter(d, x > 0.5)
))
```

```
# A tibble: 4 × 6
  expression                   min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>              <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 d[d$x > 0.5, ]            75.8µs   90.8µs    10784.  234.56KB     42.0
2 d[which(d$x > 0.5), ]     78.2µs   94.8µs    10396.  268.33KB     78.2
3 subset(d, x > 0.5)        93.6µs  116.8µs     8267.  287.09KB     61.3
4 filter(d, x > 0.5)       249.4µs    285µs     3457.    1.47MB     25.8
```

Larger n

```r
d = tibble(
  x = runif(1e6),
  y = runif(1e6)
)

(b = bench::mark(
  d[d$x > 0.5, ],
  d[which(d$x > 0.5), ],
  subset(d, x > 0.5),
  filter(d, x > 0.5)
))
```

```
# A tibble: 4 × 6
  expression                 min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>            <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 d[d$x > 0.5, ]          7.54ms   7.99ms     125.     13.3MB     54.2
2 d[which(d$x > 0.5), ]   9.11ms   9.35ms     107.     24.8MB    112. 
3 subset(d, x > 0.5)     11.61ms  12.11ms      80.9    24.8MB     91.0
4 filter(d, x > 0.5)       9.5ms  10.37ms      94.8    24.8MB    105. 
```

bench - relative results

```r
summary(b, relative=TRUE)
```

```
# A tibble: 4 × 6
  expression                 min median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>               <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
1 d[d$x > 0.5, ]            1      1         1.55      1        1   
2 d[which(d$x > 0.5), ]     1.21   1.17      1.32      1.86     2.06
3 subset(d, x > 0.5)        1.54   1.52      1         1.86     1.68
4 filter(d, x > 0.5)        1.26   1.30      1.17      1.86     1.94
```

`t.test`

> Imagine we have run 1000 experiments (rows), each of which collects data on 50 individuals (columns). The first 25 individuals in each experiment are assigned to group 1 and the rest to group 2.

The goal is to calculate the t-statistic for each experiment comparing group 1 to group 2.

```r
m = 1000
n = 50
X = matrix(
  rnorm(m * n, mean = 10, sd = 3),
  ncol = m
) |>
  as.data.frame() |>
  set_names(paste0("exp", seq_len(m))) |>
  mutate(
    ind = seq_len(n),
    group = rep(1:2, each = n/2)
  ) |>
  as_tibble() |>
  relocate(ind, group)
```

```r
X
```

```
# A tibble: 50 × 1,002
     ind group  exp1  exp2  exp3  exp4  exp5
exp6
   <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl>
<dbl>
 1     1     1  2.83  7.90  8.39 10.5  11.5
11.5
 2     2     1 14.5   8.59 11.8   9.75  9.34
4.43
 3     3     1  8.74  6.79  7.33  8.43  7.66
11.5
 4     4     1 13.1   5.51  6.39  9.67  8.55
16.3
 5     5     1  4.51 13.1  13.4   4.57  2.89
10.4
 6     6     1 11.3   5.18 12.2   5.59 10.7
12.2
 7     7     1  9.65  8.53  5.32 11.8  11.5
```

Implementations

```r
ttest_formula = function(X, m) {
  for(i in 1:m) t.test(X[[2+i]] ~ X$group)$stat
}
system.time(ttest_formula(X,m))
```

```
 user  system elapsed 
0.143   0.005   0.148 
```

```r
ttest_for = function(X, m) {
  for(i in 1:m) t.test(X[[2+i]][X$group == 1], X[[2+i]][X$group == 2])$stat
}
system.time(ttest_for(X,m))
```

```
 user  system elapsed 
0.044   0.000   0.044 
```

```r
ttest_apply = function(X) {
  f = function(x, g) {
    t.test(x[g==1], x[g==2])$stat
  }
  apply(X[,-(1:2)], 2, f, X$group)
}
system.time(ttest_apply(X))
```

```
 user  system elapsed 
0.037   0.000   0.037 
```

Implementations (cont.)

```r
ttest_hand_calc = function(X) {
  f = function(x, grp) {
    t_stat = function(x) {
      m = mean(x)
      n = length(x)
      var = sum((x - m) ^ 2) / (n - 1)

      list(m = m, n = n, var = var)
    }

    g1 = t_stat(x[grp == 1])
    g2 = t_stat(x[grp == 2])

    se_total = sqrt(g1$var / g1$n + g2$var / g2$n)
    (g1$m - g2$m) / se_total
  }

    apply(X[,-(1:2)], 2, f, X$group)
}
system.time(ttest_hand_calc(X))
```

```
   user  system elapsed 
  0.010   0.000   0.011 
```

Comparison

```r
bench::mark(
  ttest_formula(X, m),
  ttest_for(X, m),
  ttest_apply(X),
  ttest_hand_calc(X),
  check=FALSE
)
```

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

```
# A tibble: 4 × 6
  expression               min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>          <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 ttest_formula(X, m) 137.04ms  139.3ms      7.14    8.24MB     14.3
2 ttest_for(X, m)      46.77ms  53.69ms     19.1     1.91MB     15.3
3 ttest_apply(X)       39.74ms  45.43ms     21.9     3.48MB     14.0
4 ttest_hand_calc(X)    6.35ms   6.66ms    127.      3.44MB     15.8
```

Parallelization

`parallel`

Part of the base packages in R

- tools for the forking of R processes (some functions do not work on Windows)
- Core functions:
    - `detectCores`
    - `pvec`
    - `mclapply`
    - `mcparallel` & `mccollect`

`detectCores`

Surprisingly, detects the number of cores of the current system.

```
detectCores()
```

```
[1] 14
```

pvec

Parallelization of a vectorized function call

```r
system.time(pvec(1:1e7, sqrt, mc.cores = 1))
```

```
 user  system elapsed 
0.011   0.007   0.018 
```

```r
system.time(pvec(1:1e7, sqrt, mc.cores = 4))
```

```
 user  system elapsed 
0.211   0.876   1.005 
```

```r
system.time(pvec(1:1e7, sqrt, mc.cores = 8))
```

```
 user  system elapsed 
0.087   0.842   0.816 
```

```r
system.time(sqrt(1:1e7))
```

```
 user  system elapsed 
0.012   0.012   0.026 
```

pvec - `bench::system_time`

```r
bench::system_time(pvec(1:1e7, sqrt, mc.cores = 1))
```

```
process    real
 22.1ms  21.9ms
```

```r
bench::system_time(pvec(1:1e7, sqrt, mc.cores = 4))
```

```
process    real
  482ms   802ms
```

```r
bench::system_time(pvec(1:1e7, sqrt, mc.cores = 8))
```

```
process    real
  551ms   835ms
```

```r
bench::system_time(Sys.sleep(.5))
```

```
process    real
   80µs   497ms
```

```r
system.time(Sys.sleep(.5))
```

```
   user  system elapsed
  0.000   0.000   0.505
```

Cores by size

```r
cores = c(1,4,6,8,10)
order = 6:8
f = function(x,y) {
  system.time(
    pvec(1:(10^y), sqrt, mc.cores = x)
  )[3]
}

res = map(
  cores,
  function(x) {
     map_dbl(order, f, x = x)
  }
) |>
  do.call(rbind, args = _)

rownames(res) = paste0(cores," cores")
colnames(res) = paste0("10^",order)
```

```r
res
```

```
          10^6  10^7  10^8
1 cores  0.003 0.026 0.348
4 cores  0.093 0.736 8.363
6 cores  0.085 0.762 7.685
8 cores  0.105 0.760 7.987
10 cores 0.117 0.785 7.915
```

mclapply

implements a parallelized version of `lapply`

```r
system.time(rnorm(1e7))
```

```
 user  system elapsed 
0.158   0.004   0.161 
```

```r
system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 2)))
```

```
 user  system elapsed 
0.202   0.770   0.851 
```

```r
system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 4)))
```

```
 user  system elapsed 
0.206   0.744   0.764 
```

```r
system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 8)))
```

```
 user  system elapsed 
0.212   0.799   0.778 
```

```r
system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 10)))
```

```
 user  system elapsed 
0.210   0.794   0.768 
```

mcparallel

Asynchronously evaluation of an R expression in a separate process

```r
m = mcparallel(rnorm(1e6))
n = mcparallel(rbeta(1e6,1,1))
o = mcparallel(rgamma(1e6,1,1))
```

```r
str(m)
```

```
List of 2
 $ pid: int 33195
 $ fd : int [1:2] 5 8
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```r
str(n)
```

```
List of 2
 $ pid: int 33196
 $ fd : int [1:2] 6 10
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

mccollect

Checks `mcparallel` objects for completion

```r
str(mccollect(list(m,n,o)))
```

```
List of 3
 $ 33195: num [1:1000000] 0.12 0.987 -0.498 -1.111 -1.084 ...
 $ 33196: num [1:1000000] 0.997 0.082 0.139 0.198 0.649 ...
 $ 33197: num [1:1000000] 0.594 0.291 1.73 1.022 0.347 ...
```

mccollect - waiting

```r
p = mcparallel(mean(rnorm(1e5)))
```

```r
mccollect(p, wait = FALSE, 10)
```

```
$`33198`
[1] 0.002961617
```

```r
mccollect(p, wait = FALSE)
```

```
Warning in selectChildren(jobs, timeout): cannot wait for child 33198
as it does not exist
```

```
NULL
```

```r
mccollect(p, wait = FALSE)
```

```
Warning in selectChildren(jobs, timeout): cannot wait for child 33198
as it does not exist
```

```
NULL
```

doMC & foreach

doMC & foreach

Packages by Revolution Analytics that provides the `foreach` function which is a parallelizable `for` loop (and then some).

- Core functions:
  - `registerDoMC`
  - `foreach`, `%dopar%`, `%do%`

`registerDoMC`

Primarily used to set the number of cores used by `foreach`, by default uses `options("cores")` or half the number of cores found by `detectCores` from the parallel package.

```r
options("cores")
```

```
$cores
NULL
```

```r
detectCores()
```

```
[1] 14
```

```r
getDoParWorkers()
```

```
[1] 1
```

```r
registerDoMC(4)
getDoParWorkers()
```

```
[1] 4
```

`foreach`

A slightly more powerful version of base `for` loops (think `for` with an `lapply` flavor). Combined with `%do%` or `%dopar%` for single or multicore execution.

```r
for(i in 1:10) {
  sqrt(i)
}
```

```r
foreach(i = 1:5) %do% {
  sqrt(i)
}
```

```
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068
```

`foreach` - iterators

`foreach` can iterate across more than one value, but it doesn't do length coercion

```r
foreach(i = 1:5, j = 1:5) %do% {
  sqrt(i^2+j^2)
}
```

```
[[1]]
[1] 1.414214

[[2]]
[1] 2.828427

[[3]]
[1] 4.242641

[[4]]
[1] 5.656854

[[5]]
[1] 7.071068
```

```r
foreach(i = 1:5, j = 1:2) %do% {
  sqrt(i^2+j^2)
}
```

```
[[1]]
[1] 1.414214

[[2]]
[1] 2.828427
```

foreach - combining results

```
foreach(i = 1:5, .combine='c') %do% {
  sqrt(i)
}
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
foreach(i = 1:5, .combine='cbind') %do% {
  sqrt(i)
}
```

```
     result.1 result.2 result.3 result.4 result.5
[1,]        1 1.414214 1.732051        2 2.236068
```

```
foreach(i = 1:5, .combine='+') %do% {
  sqrt(i)
}
```

```
[1] 8.382332
```

`foreach` - parallelization

Swapping out `%do%` for `%dopar%` will use the parallel backend.

```r
registerDoMC(4)
system.time(foreach(i = 1:10) %dopar% mean(rnorm(1e6)))
```

```
 user  system elapsed 
0.124   0.035   0.080 
```

```r
registerDoMC(8)
system.time(foreach(i = 1:10) %dopar% mean(rnorm(1e6)))
```

```
 user  system elapsed 
0.149   0.044   0.061 
```

```r
registerDoMC(10)
system.time(foreach(i = 1:10) %dopar% mean(rnorm(1e6)))
```

```
 user  system elapsed 
0.186   0.050   0.050 
```

furrr

www.rstudio.com

furrr / future

```r
system.time( purrr::map(c(1,1,1), Sys.sleep) )
```

```
 user  system elapsed 
0.000   0.000   3.012 
```

```r
system.time( furrr::future_map(c(1,1,1), Sys.sleep) )
```

```
 user  system elapsed 
0.032   0.008   3.074 
```

```r
future::plan(future::multisession) # See also future::multicore
system.time( furrr::future_map(c(1,1,1), Sys.sleep) )
```

```
 user  system elapsed 
0.188   0.007   1.314 
```

Example - Bootstraping

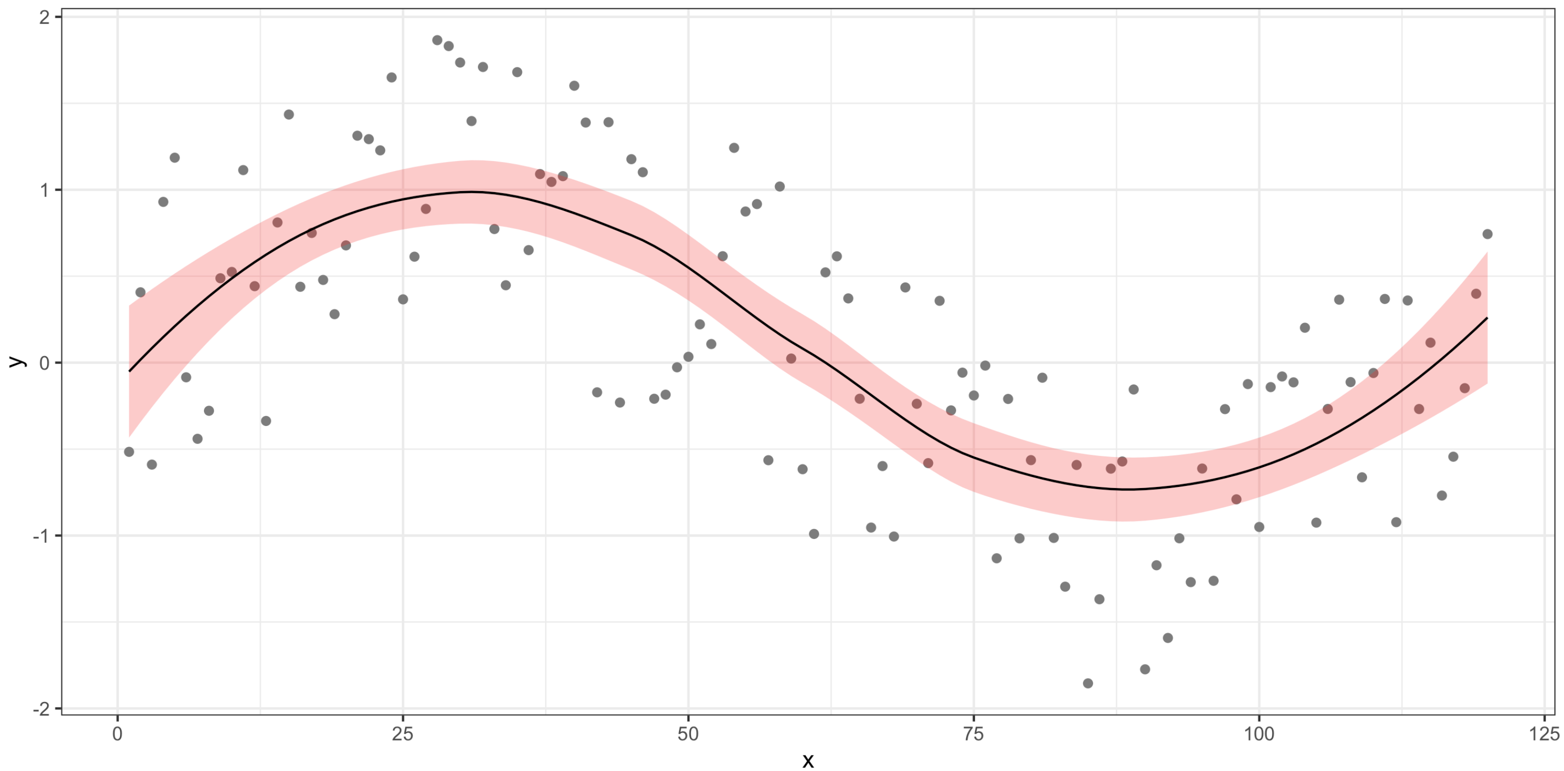
Bootstrapping is a resampling scheme where the original data is repeatedly reconstructed by taking a samples of size *n* (with replacement) from the original data, and using that to repeat an analysis procedure of interest. Below is an example of fitting a local regression (`loess`) to some synthetic data, we will construct a bootstrap prediction interval for this model.

```r
set.seed(3212016)
d = data.frame(x = 1:120) |>
    mutate(y = sin(2*pi*x/120) + runif(length(x),-1,1))

l = loess(y ~ x, data=d)
p = predict(l, se=TRUE)

d = d |> mutate(
  pred_y = p$fit,
  pred_y_se = p$se.fit
)
```

```r
ggplot(d, aes(x,y)) +
  geom_point(color="gray50") +
  geom_ribbon(
    aes(ymin = pred_y - 1.96 * pred_y_se,
        ymax = pred_y + 1.96 * pred_y_se),
    fill="red", alpha=0.25
  ) +
  geom_line(aes(y=pred_y)) +
  theme_bw()
```



Bootstraping Demo

What to use when?

Optimal use of parallelization / multiple cores is hard, there isn’t one best solution

- Don’t underestimate the overhead cost
- Experimentation is key
- Measure it or it didn’t happen
- Be aware of the trade off between developer time and run time

BLAS and LAPACK

Statistics and Linear Algebra

An awful lot of statistics is at its core linear algebra.

For example:

- Linear regession models, find

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

- Principle component analysis
  - Find $T = XW$ where W is a matrix whose columns are the eigenvectors of $X^T X$.
  - Often solved via SVD - Let $X = U\Sigma W^T$ then $T = U\Sigma$.

Numerical Linear Algebra

Not unique to Statistics, these are the type of problems that come up across all areas of numerical computing.

- Numerical linear algebra $\neq$ mathematical linear algebra
- Efficiency and stability of numerical algorithms matter
  - Designing and implementing these algorithms is hard
- Don't reinvent the wheel - common core linear algebra tools (well defined API)

BLAS and LAPACK

Low level algorithms for common linear algebra operations

BLAS

- **B**asic **L**inear **A**lgebra **S**ubprograms
- Copying, scaling, multiplying vectors and matrices
- Origins go back to 1979, written in Fortran

LAPACK

- **L**inear **A**lgebra **Pack**age
- Higher level functionality building on BLAS.
- Linear solvers, eigenvalues, and matrix decompositions
- Origins go back to 1992, mostly Fortran (expanded on LINPACK, EISPACK)

Modern variants?

Most default BLAS and LAPACK implementations (like R’s defaults) are somewhat dated

- Written in Fortran and designed for a single cpu core
- Certain (potentially non-optimal) hard coded defaults (e.g. block size).

Multithreaded alternatives:

- ATLAS - Automatically Tuned Linear Algebra Software
- OpenBLAS - fork of GotoBLAS from TACC at UTexas
- Intel MKL - Math Kernel Library, part of Intel’s commercial compiler tools
- cuBLAS / Magma - GPU libraries from Nvidia and UTK respectively
- Accelerate / vecLib - Apple’s framework for GPU and multicore computing

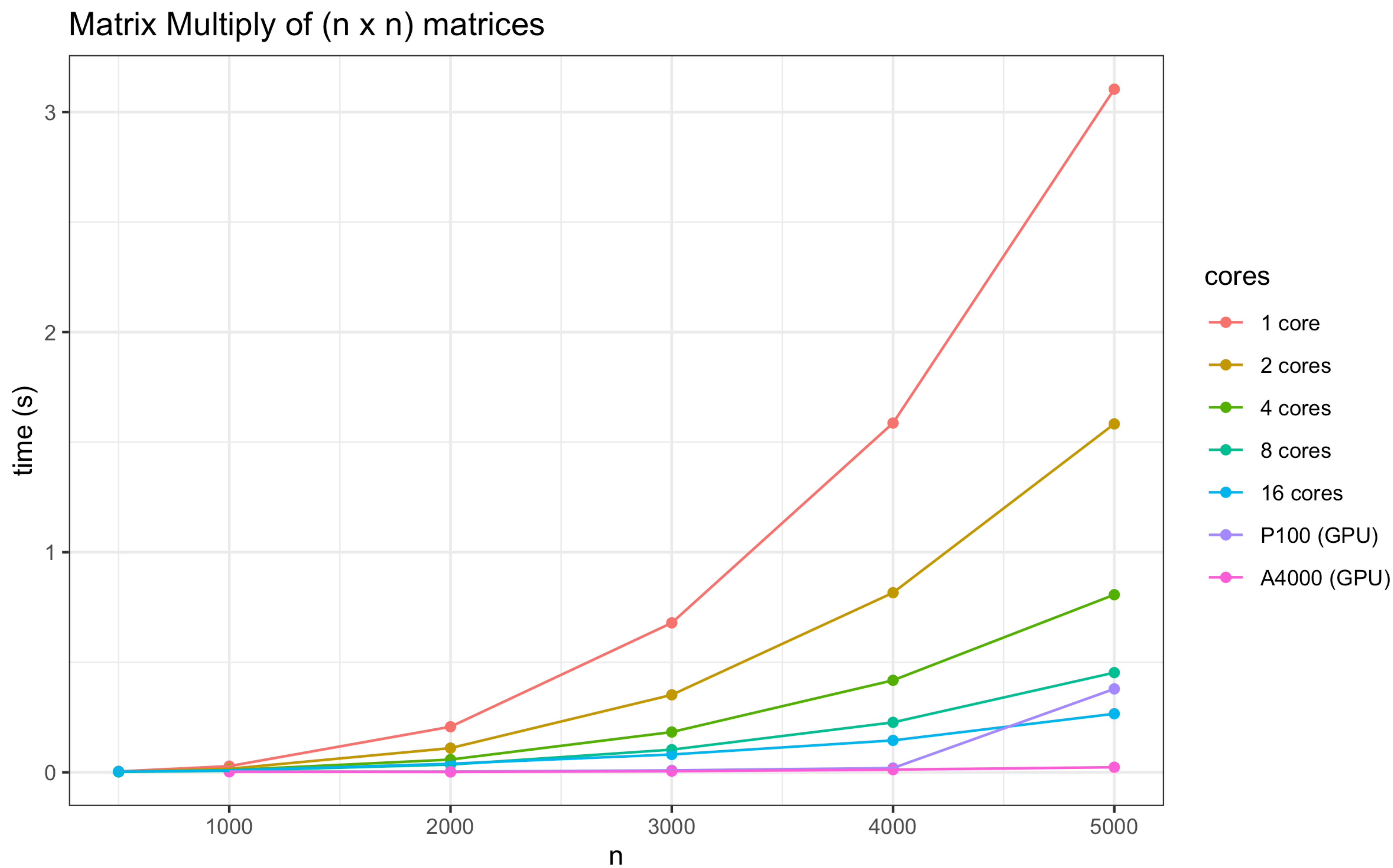
OpenBLAS Matrix Multiply Performance

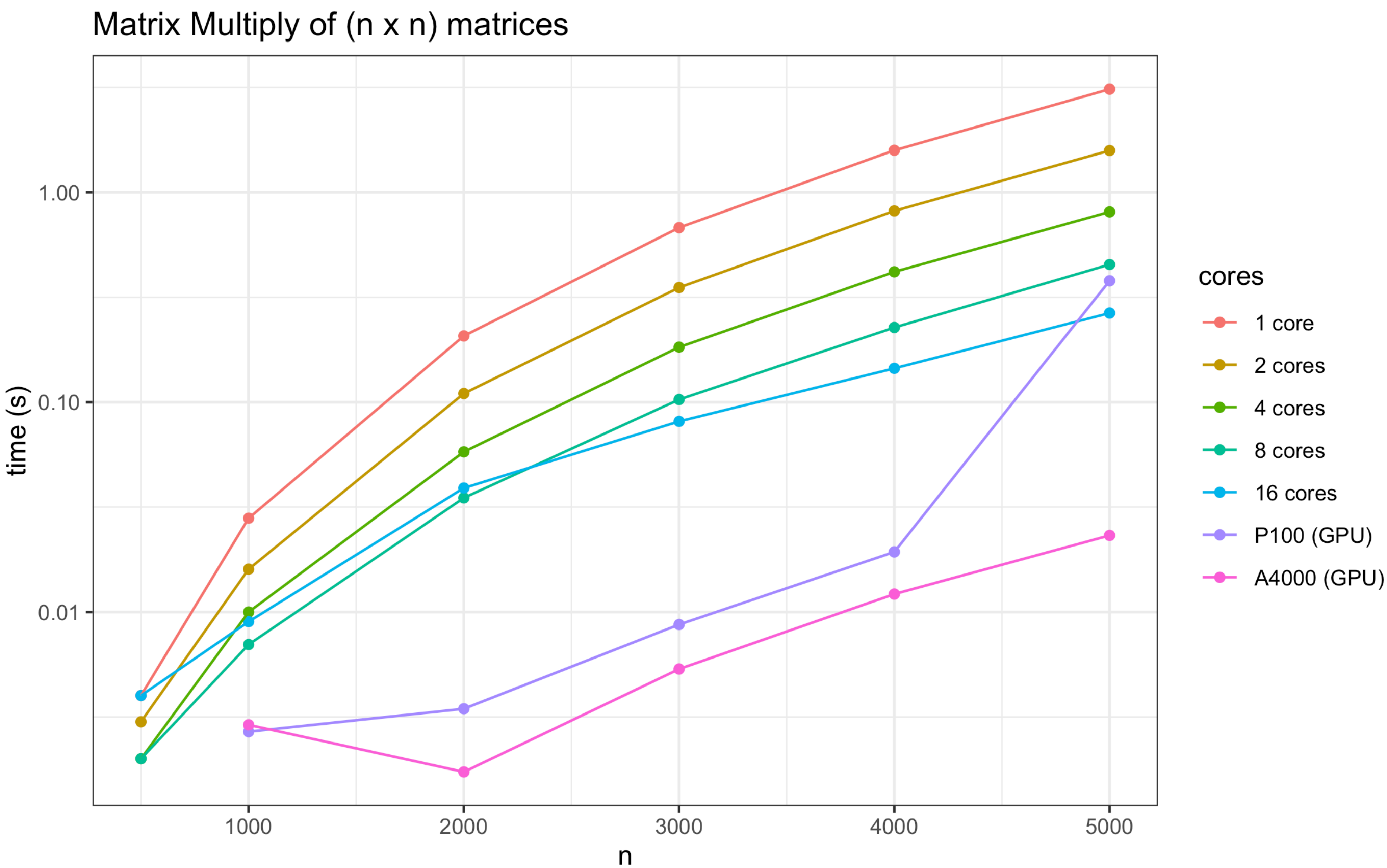
```r
x=matrix(runif(5000^2),ncol=5000)

sizes = c(100,500,1000,2000,3000,4000,5000)
cores = c(1,2,4,8,16)

sapply(
  cores,
  function(n_cores) {
    flexiblas::flexiblas_set_num_threads(n_cores)
    sapply(
      sizes,
      function(s) {
        y = x[1:s,1:s]
        system.time(y %*% y)[3]
      }
    )
  }
)
```

n	1 core	2 cores	4 cores	8 cores	16 cores
100	0.000	0.000	0.000	0.000	0.000
500	0.004	0.003	0.002	0.002	0.004
1000	0.028	0.016	0.010	0.007	0.009
2000	0.207	0.110	0.058	0.035	0.039
3000	0.679	0.352	0.183	0.103	0.081
4000	1.587	0.816	0.418	0.227	0.145
5000	3.104	1.583	0.807	0.453	0.266


Matrix Multiply of (n x n) matrices
time (s)
3
2
1
0
1000
2000
3000
4000
5000
n
cores
1 core
2 cores
4 cores
8 cores
16 cores
P100 (GPU)
A4000 (GPU)


Matrix Multiply of (n x n) matrices
time (s)
1.00
0.10
0.01
1000
2000
3000
4000
5000
n
cores
1 core
2 cores
4 cores
8 cores
16 cores
P100 (GPU)
A4000 (GPU)