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

```r
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$x > 0.5, ]           39.3μs   43.9μs    21703.  235.44KB     60.4
2 d[which(d$x > 0.5), ]    55.1μs   58.9μs    16690.  269.94KB     86.5
3 subset(d, x > 0.5)       63.5μs   67.9μs    14549.  288.01KB     73.8
4 filter(d, x > 0.5)      303.9μs  327.3μs     2960.    1.47MB     16.7
```

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$x > 0.5, ]           2.81ms    2.9ms      315.    13.4MB     242.
2 d[which(d$x > 0.5), ]    6.03ms   6.18ms      147.    24.8MB     264.
3 subset(d, x > 0.5)       6.75ms   6.91ms      129.    24.8MB     249.
4 filter(d, x > 0.5)       4.13ms   4.38ms      186.    24.8MB     299.
```

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$x > 0.5, ]           1      1         2.45      1        1   
2 d[which(d$x > 0.5), ]  2.15   2.13      1.14      1.86     1.09
3 subset(d, x > 0.5)     2.41   2.38      1         1.86     1.03
4 filter(d, x > 0.5)     1.47   1.51      1.45      1.86     1.24
```

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  8.66 10.5   5.83  5.35 10.9  12.8 
 2     2     1 17.0  13.8  14.2   5.30  7.47 10.5 
 3     3     1  6.54  9.09 14.3  10.7  11.4   8.26
 4     4     1  7.04 11.7  14.2   5.76 10.1   1.50
 5     5     1  8.41  5.29  8.96 13.1   8.49  7.66
 6     6     1 11.0   6.49  6.44  9.23  9.80  9.62
 7     7     1  8.15 10.2  11.6  12.0   6.85  8.03
 8     8     1 11.4  12.3   9.82  8.48 11.2   8.08
 9     9     1 12.8   9.49 12.6  17.3   8.99  9.11
10    10     1  9.60  6.23 11.7  11.8   8.27 11.0 
# ℹ 40 more rows
# ℹ 994 more variables: exp7 <dbl>, exp8 <dbl>,
#   exp9 <dbl>, exp10 <dbl>, exp11 <dbl>,
#   exp12 <dbl>, exp13 <dbl>, exp14 <dbl>,
#   exp15 <dbl>, exp16 <dbl>, exp17 <dbl>,
#   exp18 <dbl>, exp19 <dbl>, exp20 <dbl>,
```

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 
0.197   0.000   0.198 
```

```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 
0.063   0.000   0.063 
```

```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 
0.054   0.000   0.055 
```

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 
  0.016   0.000   0.016 
```

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) 218.73ms  224.1ms      4.47    8.24MB     23.8
2 ttest_for(X, m)      69.59ms   73.2ms     13.7     1.91MB     25.4
3 ttest_apply(X)       63.28ms   68.4ms     14.7     3.48MB     25.8
4 ttest_hand_calc(X)    9.36ms   10.1ms     72.4     3.44MB     21.5
```

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.

```r
detectCores()
```

```
[1] 32
```

pvec

Parallelization of a vectorized function call

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

```
 user  system elapsed 
0.028   0.038   0.068 
```

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

```
 user  system elapsed 
0.211   0.253   0.324 
```

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

```
 user  system elapsed 
0.101   0.288   0.176 
```

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

```
 user  system elapsed 
0.024   0.042   0.066 
```

pvec - bench::system_time

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

```
process    real
  125ms   126ms
```

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

```
process    real
  220ms   250ms
```

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

```
process    real
  272ms   286ms
```

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

```
process    real 
 54.4µs 500.8ms 
```

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

```
   user  system elapsed 
  0.000   0.000   0.501 
```

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  0.004 0.126 0.745
4 cores  0.033 0.171 2.190
6 cores  0.038 0.173 1.847
8 cores  0.045 0.183 1.827
10 cores 0.050 0.187 1.852
```

mclapply

implements a parallelized version of `lapply`

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

```
   user  system elapsed
  0.197   0.020   0.219
```

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

```
   user  system elapsed
  0.247   0.200   0.260
```

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

```
   user  system elapsed
  0.231   0.216   0.172
```

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

```
   user  system elapsed
  0.238   0.271   0.149
```

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

```
   user  system elapsed
  0.246   0.332   0.144
```

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 787800
 $ fd : int [1:2] 6 9
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```r
str(n)
```

```
List of 2
 $ pid: int 787801
 $ fd : int [1:2] 7 11
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

mccollect

Checks `mcparallel` objects for completion

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

```
List of 3
 $ 787800: num [1:1000000] 1.113 -1.375 0.254 -1.055 0.641 ...
 $ 787801: num [1:1000000] 0.5526 0.0744 0.9768 0.9385 0.1238 ...
 $ 787802: num [1:1000000] 2.00498 4.60403 0.00115 0.58452 0.48502 ...
```

mccollect - waiting

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

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

```
$`787803`
[1] -0.0008566918
```

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

```
Warning in selectChildren(jobs, timeout): cannot wait for child
787803 as it does not exist
```

```
NULL
```

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

```
Warning in selectChildren(jobs, timeout): cannot wait for child
787803 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] 32
```

```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

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

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

```r
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
```

```r
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 
  0.164   0.051   0.090 
```

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

```
   user  system elapsed 
  0.176   0.098   0.063 
```

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

```
   user  system elapsed 
  0.202   0.139   0.070 
```

furrr
www.rstudio.com

furrr / future

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

```
 user  system elapsed 
0.000   0.000   3.003 
```

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

```
 user  system elapsed 
0.040   0.003   3.064 
```

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

```
 user  system elapsed 
0.392   0.009   1.700 
```

Example - Bootstraping

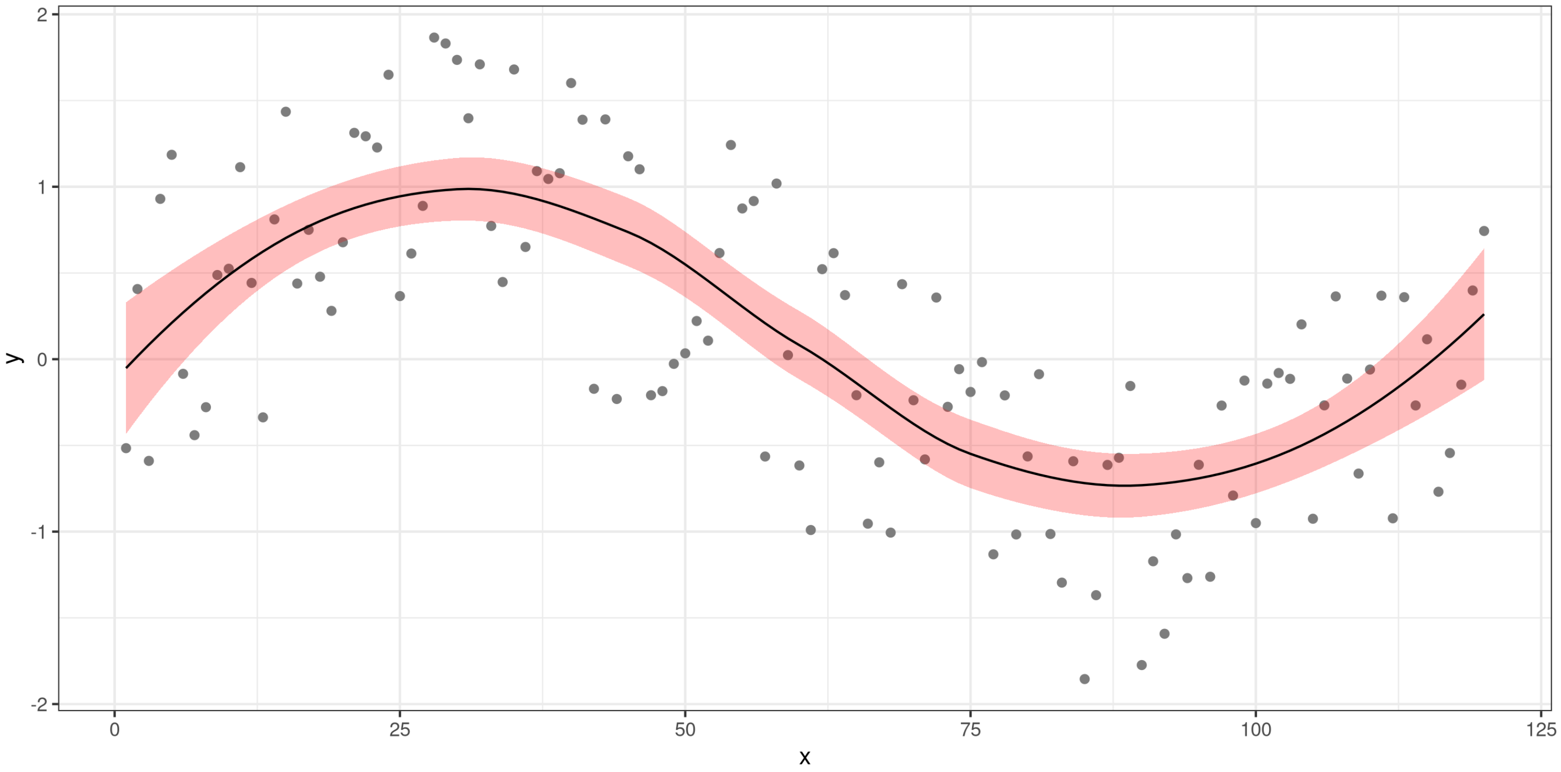
Bootstrapping is a resampling scheme where the original data is repeatedly reconstructed by taking a samples of size *n* (with replacement) from the original data, and using that to repeat an analysis procedure of interest. Below is an example of fitting a local regression (`loess`) to some synthetic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

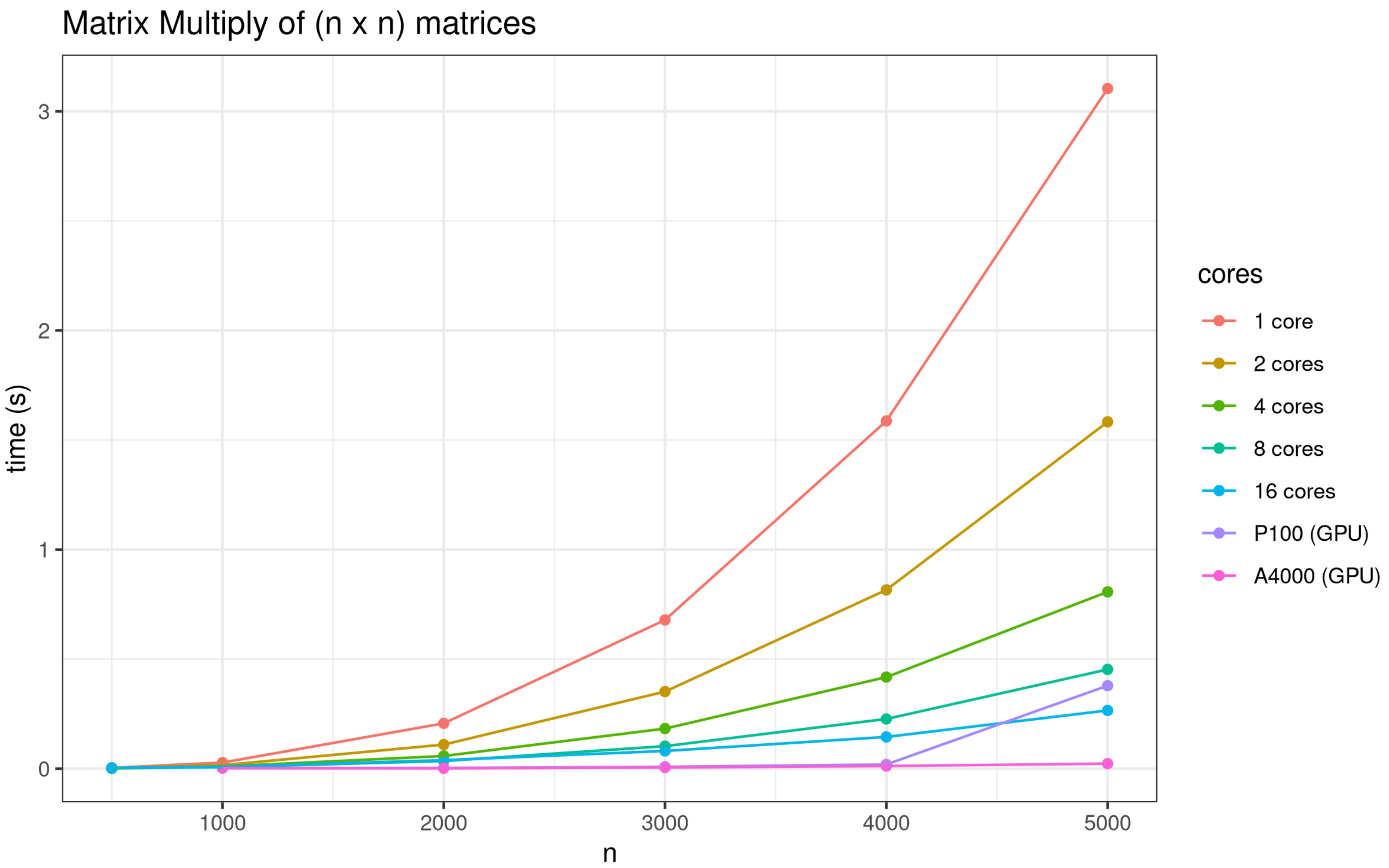
OpenBLAS Matrix Multiply Performance

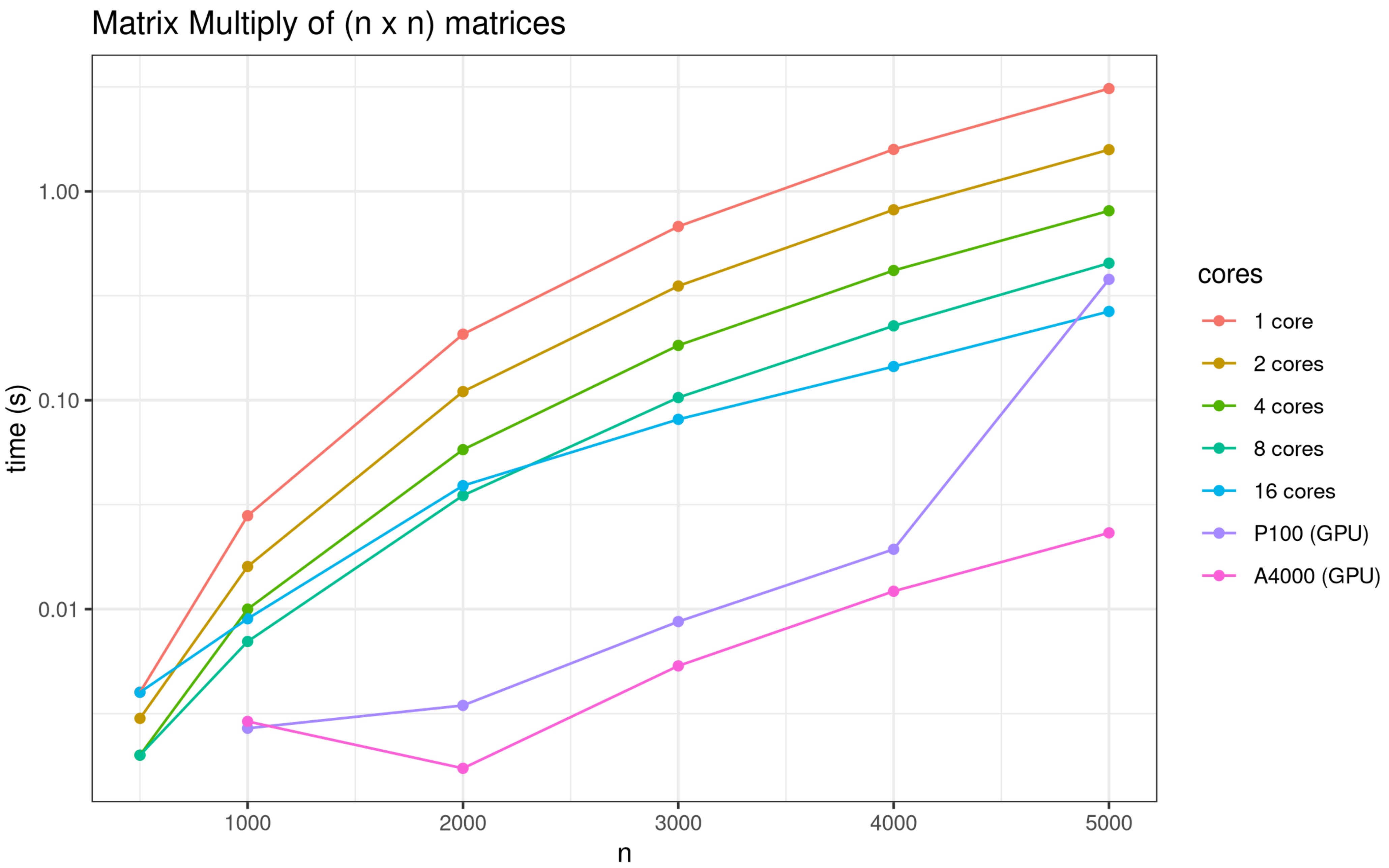
```r
x=matrix(runif(5000^2),ncol=5000)

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
n
1000
2000
3000
4000
5000
cores
1 core
2 cores
4 cores
8 cores
16 cores
P100 (GPU)
A4000 (GPU)