Profiling & Parallelization

Lecture 20

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))
```

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$x > 0.5, ]              49.4µs   56.7µs    17251.  238.11KB     46.9
2 d[which(d$x > 0.5), ]       90.7µs    103µs     9595.  268.38KB     48.3
3 subset(d, x > 0.5)          87.5µs  106.8µs     9234.  296.33KB     49.5
4 filter(d, x > 0.5)           287µs    314µs     3061.    1.48MB     18.4
```

Larger n

```
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$x > 0.5, ]           3.62ms   4.08ms      247.    13.4MB     142.
2 d[which(d$x > 0.5), ]    8.12ms   8.93ms      111.    24.8MB     140.
3 subset(d, x > 0.5)       9.46ms  10.12ms       99.0   24.8MB     120.
4 filter(d, x > 0.5)       4.81ms   5.36ms      184.    24.8MB     272.
```

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$x > 0.5, ]            1      1         2.49      1        1.19
2 d[which(d$x > 0.5), ]  2.24   2.19      1.12      1.86     1.17
3 subset(d, x > 0.5)     2.62   2.48      1         1.86     1   
4 filter(d, x > 0.5)     1.33   1.32      1.86      1.86     2.27
```

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] 10
```

pvec

Parallelization of a vectorized function call

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

```
 user  system elapsed 
0.089   0.012   0.100 
```

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

```
 user  system elapsed 
0.126   0.119   0.198 
```

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

```
 user  system elapsed 
0.092   0.165   0.168 
```

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

```
 user  system elapsed 
0.018   0.018   0.038 
```

pvec - `bench::system_time`

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

```
process    real 
 61.4ms  60.7ms 
```

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

```
process    real 
  157ms   188ms 
```

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

```
process    real 
  175ms   207ms 
```

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

```
process    real 
   57µs   497ms 
```

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

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

Cores by size

```
cores = c(1,4,8,16)
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)

res
```

```
          10^6  10^7  10^8
1 cores  0.005 0.025 0.327
4 cores  0.041 0.161 1.687
8 cores  0.047 0.174 1.361
16 cores 0.055 0.185 1.359
```

mclapply

Parallelized version of `lapply`

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

```
 user  system elapsed 
0.267   0.006   0.275 
```

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

```
 user  system elapsed 
0.320   0.106   0.273 
```

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

```
 user  system elapsed 
0.326   0.100   0.179 
```

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

```
 user  system elapsed 
0.340   0.135   0.180 
```

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

```
   user  system elapsed 
  0.354   0.159   0.204 
```

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))

str(m)
```

```
List of 2
 $ pid: int 80672
 $ fd : int [1:2] 4 7
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```r
str(n)
```

```
List of 2
 $ pid: int 80673
 $ fd : int [1:2] 5 9
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

mccollect

Checks `mcparallel` objects for completion

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

```
List of 3
 $ 80672: num [1:1000000] -1.049 -1.028 0.705 -0.17 1.734 ...
 $ 80673: num [1:1000000] 0.373 0.989 0.259 0.713 0.649 ...
 $ 80674: num [1:1000000] 0.5921 1.2226 0.0756 2.4405 0.0465 ...
```

mccollect - waiting

```r
p = mcparallel(mean(rnorm(1e5)))
mccollect(p, wait = FALSE, 10) # will retrieve the result (since it's fast)
```

```
$`80675`
[1] 0.001344587
```

```r
mccollect(p, wait = FALSE)     # will signal the job as terminating
```

```
NULL
```

```r
mccollect(p, wait = FALSE)     # there is no longer such a job
```

```
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.

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

```
$cores
NULL
```

```
detectCores()
```

```
[1] 10
```

```
getDoParWorkers()
```

```
[1] 1
```

```
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)
}

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

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

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

```
[[4]]
[1] 2
```

`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]]
```

```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)
```

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

```
 user  system elapsed 
0.296   0.027   0.110 
```

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

```
 user  system elapsed 
0.317   0.045   0.082 
```

```r
registerDoMC(12)
system.time(foreach(i = 1:10) %dopar% mean(rnorm(1e6)))
```

```
 user  system elapsed 
0.329   0.047   0.074 
```

furrr

www.rstudio.com

furrr / future

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

```
 user  system elapsed 
0.001   0.000   3.012 
```

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

```
 user  system elapsed 
0.050   0.007   3.083 
```

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

```
 user  system elapsed 
0.163   0.004   1.454 
```

Example - Bootstraping

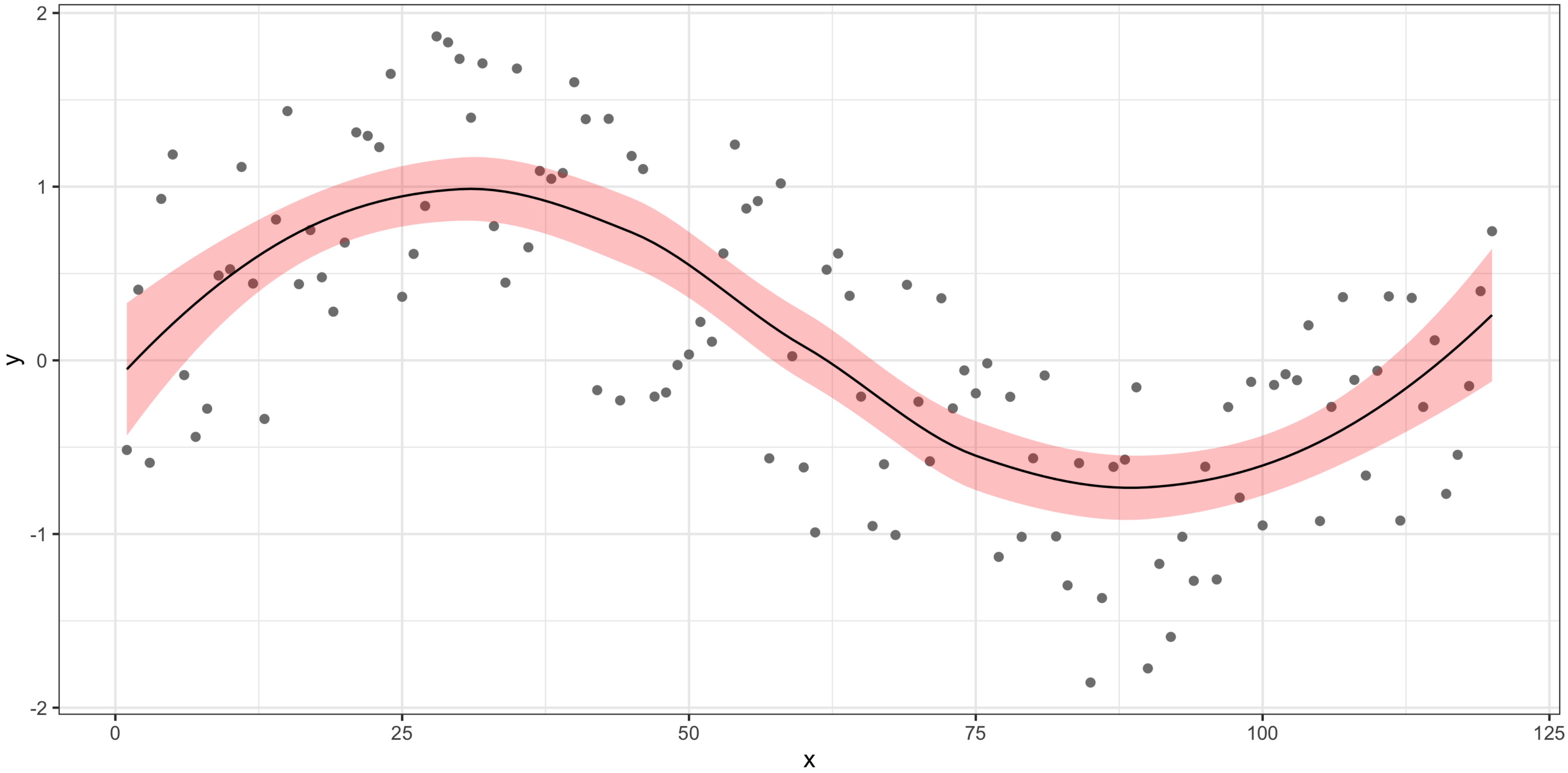
Bootstrapping is a resampling scheme where the original data is repeatedly reconstructed by taking a samples of size *n* (with replacement) from the original data, and using that to repeat an analysis procedure of interest. Below is an example of fitting a local regression (`loess`) to some synthetic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

```
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()
```



2
1
0
-1
-2
y
0
25
50
75
100
125
x

What to use when?

Optimal use of 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

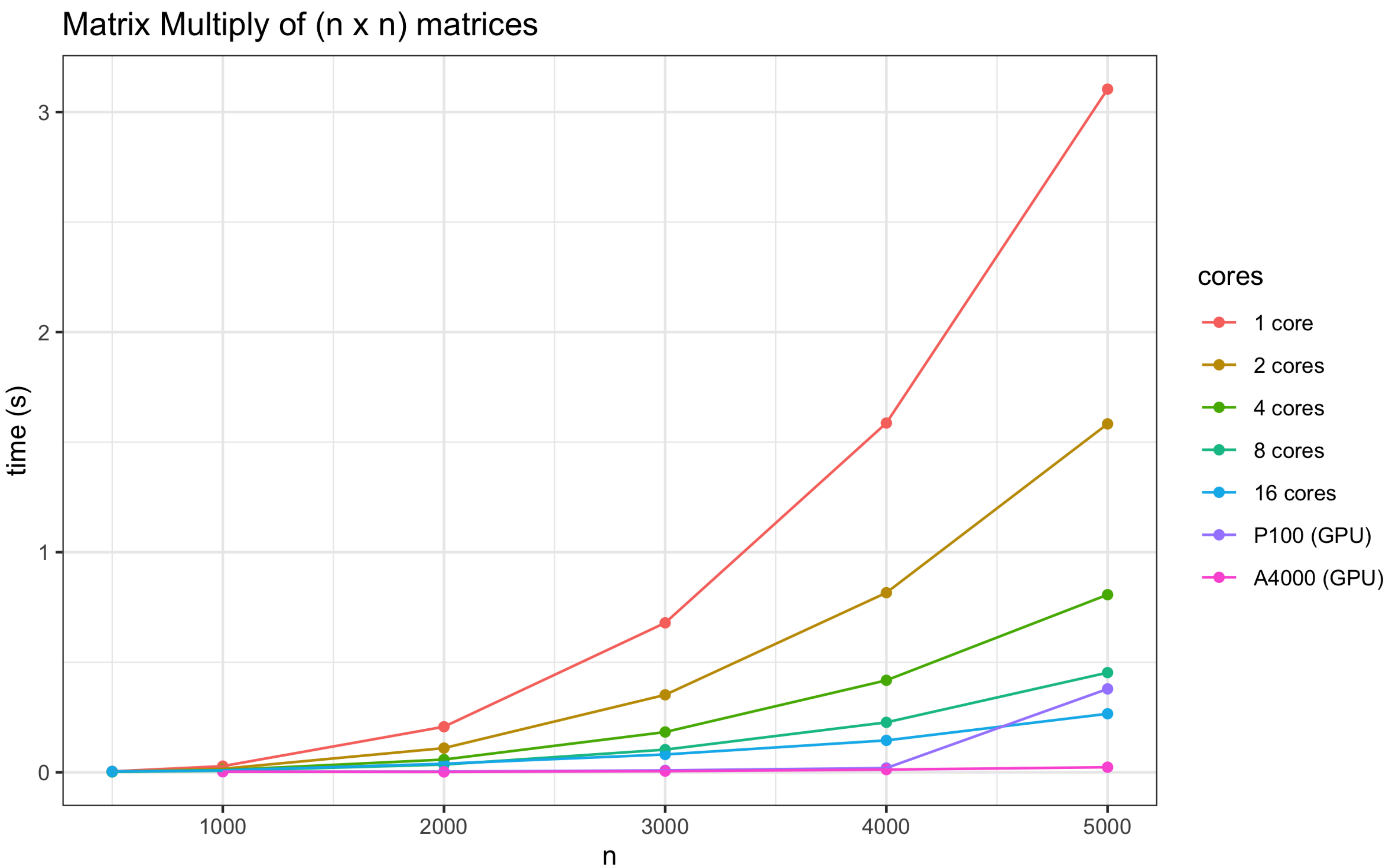
OpenBLAS Matrix Multiply Performance

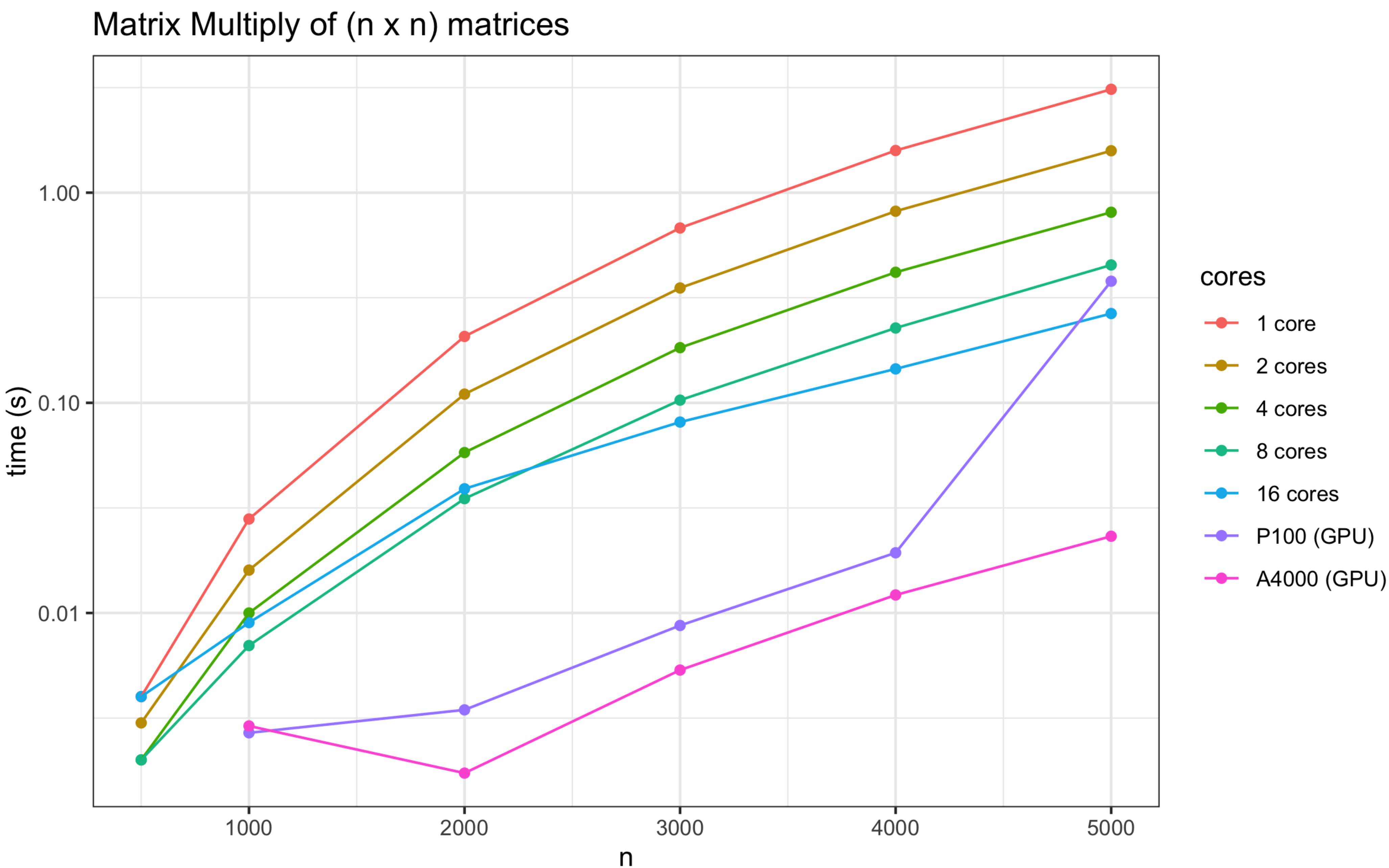
```r
x=matrix(runif(5000^2),ncol=5000)

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)