

Profiling & Parallelization

Lecture 21

Dr. Colin Rundel

Profiling & Benchmarking

profvis demo

```
1 n = 1e6
2 d = tibble(
3   x1 = rt(n, df = 3),
4   x2 = rt(n, df = 3),
5   x3 = rt(n, df = 3),
6   x4 = rt(n, df = 3),
7   x5 = rt(n, df = 3),
8 ) |>
9   mutate(y = -2*x1 - 1*x2 + 0*x3 + 1*x4 + 2*x5 + rnorm(n))
```

```
1 profvis::profvis(lm(y~., data=d))
```

profvis demo 2

```
1 profvis::profvis({  
2   data = data.frame(value = runif(5e4))  
3  
4   data$sum[1] = data$value[1]  
5   for (i in seq(2, nrow(data))) {  
6     data$sum[i] = data$sum[i-1] + data$value[i]  
7   }  
8 })
```

```
1 profvis::profvis({  
2   x = runif(5e4)  
3   sum = x[1]  
4   for (i in seq(2, length(x))) {  
5     sum[i] = sum[i-1] + x[i]  
6   }  
7 })
```

Benchmarking - bench

```
1 d = tibble(  
2   x = runif(10000),  
3   y = runif(10000)  
4 )  
5  
6 (b = bench::mark(  
7   d[d$x > 0.5, ],  
8   d[which(d$x > 0.5), ],  
9   subset(d, x > 0.5),  
10  filter(d, x > 0.5)  
11 ))
```

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,]	128µs	138µs	7179.	238.91KB	19.4
2 d[which(d\$x > 0.5),]	137µs	152µs	6514.	269.81KB	33.9
3 subset(d, x > 0.5)	168µs	190µs	5138.	287.91KB	26.3
4 filter(d, x > 0.5)	385µs	412µs	2384.	1.47MB	42.3

Larger n

```
1 d = tibble(  
2   x = runif(1e6),  
3   y = runif(1e6)  
4 )  
5  
6 (b = bench::mark(  
7   d[d$x > 0.5, ],  
8   d[which(d$x > 0.5), ],  
9   subset(d, x > 0.5),  
10  filter(d, x > 0.5)  
11 ))
```

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,]	11.6ms	12.2ms	82.5	13.4MB	78.2
2 d[which(d\$x > 0.5),]	12.8ms	13.6ms	74.0	24.8MB	123.
3 subset(d, x > 0.5)	17.3ms	17.8ms	56.1	24.8MB	84.2
4 filter(d, x > 0.5)	14ms	14.2ms	70.4	24.8MB	51.9

bench - relative results

```
1 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	1.47	1	1.51
2	d[which(d\$x > 0.5),]	1.10	1.12	1.32	1.86	2.38
3	subset(d, x > 0.5)	1.49	1.46	1	1.86	1.62
4	filter(d, x > 0.5)	1.20	1.17	1.25	1.86	1

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.

```
1 m = 1000
2 n = 50
3 X = matrix(
4   rnorm(m * n, mean = 10, sd = 3),
5   ncol = m
6 ) |>
7   as.data.frame() |>
8   set_names(paste0("exp", seq_len(m))) |>
9   mutate(
10     ind = seq_len(n),
11     group = rep(1:2, each = n/2)
12 ) |>
13   as_tibble() |>
14   relocate(ind, group)
```

```
1 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  9.54 10.9 11.8 12.2 12.2 12.6
2     2     1  7.78 12.7  9.21  9.48 13.1  7.67
3     3     1 10.5  10.8  7.36  7.04 12.1  6.23
4     4     1  8.71  7.36  9.20 12.4  4.42 12.8
5     5     1  7.38  9.47 10.5 13.4 10.2  7.35
6     6     1  9.68  9.71 10.9  8.89 10.9 13.3
7     7     1  7.40 13.2  6.02  8.76  7.02  3.30
8     8     1 12.7 14.2  3.93  8.53  5.45  6.96
9     9     1  7.15  8.81 13.8  8.28  8.64  7.75
10    10     1  8.80 14.4 15.9  7.93 12.3 13.6
# i 40 more rows
# i 994 more variables: exp7 <dbl>, exp8 <dbl>,
#   exp9 <dbl>, exp10 <dbl>, exp11 <dbl>,
#   exp12 <dbl>, exp13 <dbl>, exp14 <dbl>.
```


Implementations

```
1 ttest_formula = function(X, m) {  
2   for(i in 1:m) t.test(X[[2+i]] ~ X$group)$stat  
3 }  
4 system.time(ttest_formula(X,m))
```

```
user  system elapsed  
0.186   0.001   0.187
```

```
1 ttest_for = function(X, m) {  
2   for(i in 1:m) t.test(X[[2+i]][X$group == 1], X[[2+i]][X$group == 2])$stat  
3 }  
4 system.time(ttest_for(X,m))
```

```
user  system elapsed  
0.066   0.001   0.068
```

```
1 ttest_apply = function(X) {  
2   f = function(x, g) {  
3     t.test(x[g==1], x[g==2])$stat  
4   }  
5   apply(X[,-(1:2)], 2, f, X$group)  
6 }  
7 system.time(ttest_apply(X))
```

```
user  system elapsed  
0.051   0.001   0.052
```

Implementations (cont.)

```
1 ttest_hand_calc = function(X) {  
2   f = function(x, grp) {  
3     t_stat = function(x) {  
4       m = mean(x)  
5       n = length(x)  
6       var = sum((x - m) ^ 2) / (n - 1)  
7  
8       list(m = m, n = n, var = var)  
9     }  
10  
11     g1 = t_stat(x[grp == 1])  
12     g2 = t_stat(x[grp == 2])  
13  
14     se_total = sqrt(g1$var / g1$n + g2$var / g2$n)  
15     (g1$m - g2$m) / se_total  
16   }  
17  
18   apply(X[,-(1:2)], 2, f, X$group)  
19 }
```

user	system	elapsed
0.015	0.000	0.015

Comparison

```
1 bench::mark(  
2   ttest_formula(X, m),  
3   ttest_for(X, m),  
4   ttest_apply(X),  
5   ttest_hand_calc(X),  
6   check=FALSE  
7 )
```

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)	188.85ms	192.35ms	5.18	8.24MB	22.5
2	ttest_for(X, m)	61.45ms	64.52ms	15.6	1.91MB	23.5
3	ttest_apply(X)	54.61ms	57.83ms	17.3	3.48MB	23.0
4	ttest_hand_calc(X)	8.73ms	9.04ms	93.1	3.44MB	23.8

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`
 - `mcpipeline` & `mccollect`

detectCores

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

```
1 detectCores()
```

```
[1] 10
```

pvec

Parallelization of a vectorized function call

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

user	system	elapsed
0.017	0.012	0.028

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

user	system	elapsed
0.176	0.174	0.278

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

user	system	elapsed
0.090	0.201	0.178

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

user	system	elapsed
0.017	0.020	0.037

pvec - bench::system_time

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

process	real
63.2ms	62.3ms

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

process	real
163ms	188ms

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

process	real
219ms	230ms


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

process	real
142 μ s	493ms

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

user	system	elapsed
0.000	0.000	0.507

Cores by size

```
1 cores = c(1,4,6,8,10)
2 order = 6:8
3 f = function(x,y) {
4   system.time(
5     pvec(1:(10^y), sqrt, mc.cores = x)
6   )[3]
7 }
8
9 res = map(
10   cores,
11   function(x) {
12     map_dbl(order, f, x = x)
13   }
14 ) |>
15   do.call(rbind, args = _)
16
17 rownames(res) = paste0(cores," cores")
18 colnames(res) = paste0("10^",order)
```

```
1 res
```

		10 ⁶	10 ⁷	10 ⁸
1 cores	0.004	0.036	0.367	
4 cores	0.036	0.151	1.917	
6 cores	0.030	0.111	1.332	
8 cores	0.042	0.155	1.506	
10 cores	0.045	0.192	1.567	

mclapply

Parallelized version of lapply

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

```
   user  system elapsed  
0.264   0.003   0.267
```

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

```
   user  system elapsed  
0.329   0.112   0.284
```

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

```
   user  system elapsed  
0.334   0.102   0.180
```

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

```
   user  system elapsed  
0.340   0.150   0.174
```

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

```
   user  system elapsed  
0.362   0.162   0.178
```

mcpParallel

Asynchronously evaluation of an R expression in a separate process

```
1 m = mcpParallel(rnorm(1e6))  
2 n = mcpParallel(rbeta(1e6,1,1))  
3 o = mcpParallel(rgamma(1e6,1,1))
```

```
1 str(m)
```

List of 2

```
$ pid: int 78380  
$ fd : int [1:2] 5 8  
- attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```
1 str(n)
```

List of 2

```
$ pid: int 78381  
$ fd : int [1:2] 6 10  
- attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

mccollect

Checks `mcparallel` objects for completion

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

List of 3

```
$ 78380: num [1:1000000] -0.1712 -0.0412 -0.9246 -1.2699 -1.274 ...  
$ 78381: num [1:1000000] 0.201 0.838 0.289 0.192 0.722 ...  
$ 78382: num [1:1000000] 0.7425 0.5719 0.4059 0.0538 0.7154 ...
```

mccollect - waiting

```
1 p = mcpipeline(mean(rnorm(1e5)))
```

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

```
$`78383`
```

```
[1] 0.006051739
```

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

```
Warning in selectChildren(jobs, timeout): cannot wait for child 78383  
as it does not exist
```

```
NULL
```

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

```
Warning in selectChildren(jobs, timeout): cannot wait for child 78383  
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.

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

```
$cores
```

```
NULL
```

```
1 detectCores()
```

```
[1] 10
```

```
1 getDoParWorkers()
```

```
[1] 1
```

```
1 registerDoMC(4)
2 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.

```
1 for(i in 1:10) {  
2   sqrt(i)  
3 }
```

```
1 foreach(i = 1:5) %do% {  
2   sqrt(i)  
3 }
```

```
[[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

```
1 foreach(i = 1:5, j = 1:5) %do% {  
2   sqrt(i^2+j^2)  
3 }
```

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

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

```
[[3]]  
[1] 4.242641
```

```
[[4]]  
[1] 5.656854
```

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

```
1 foreach(i = 1:5, j = 1:2) %do% {  
2   sqrt(i^2+j^2)  
3 }
```

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

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

foreach - combining results

```
1 foreach(i = 1:5, .combine='c') %do% {  
2   sqrt(i)  
3 }
```

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

```
1 foreach(i = 1:5, .combine='cbind') %do% {  
2   sqrt(i)  
3 }
```

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

```
1 foreach(i = 1:5, .combine='+') %do% {  
2   sqrt(i)  
3 }
```

```
[1] 8.382332
```

foreach - parallelization

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

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

```
user  system elapsed
0.297  0.040   0.113
```

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

```
user  system elapsed
0.303  0.052   0.082
```

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

```
user  system elapsed
0.327  0.064   0.078
```



furrr / future

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

user	system	elapsed
0.001	0.001	3.012

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

user	system	elapsed
0.042	0.006	3.076

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

user	system	elapsed
0.202	0.006	1.459

Example - Bootstrapping

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.

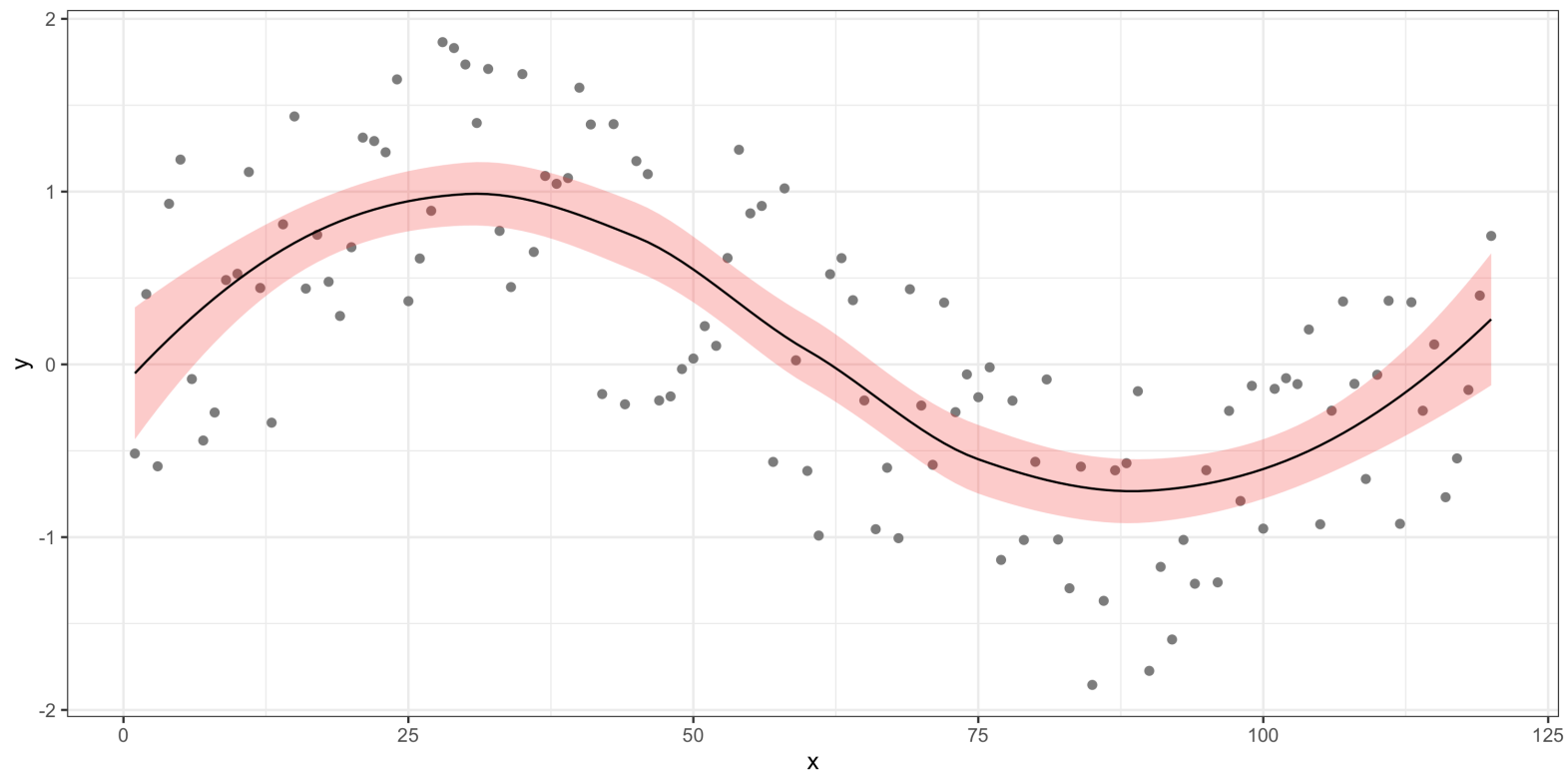
```
1  set.seed(3212016)
2  d = data.frame(x = 1:120) |>
3    mutate(y = sin(2*pi*x/120) + runif(length(x), -1, 1))
4
5  l = loess(y ~ x, data=d)
6  p = predict(l, se=TRUE)
7
8  d = d |> mutate(
9    pred_y = p$fit,
10   pred_y_se = p$se.fit
11 )
```



```

1 ggplot(d, aes(x,y)) +
2   geom_point(color="gray50") +
3   geom_ribbon(
4     aes(ymin = pred_y - 1.96 * pred_y_se,
5         ymax = pred_y + 1.96 * pred_y_se),
6     fill="red", alpha=0.25
7   ) +
8   geom_line(aes(y=pred_y)) +
9   theme_bw()

```



Bootstrapping 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 regression 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 **P**ackage
- 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:

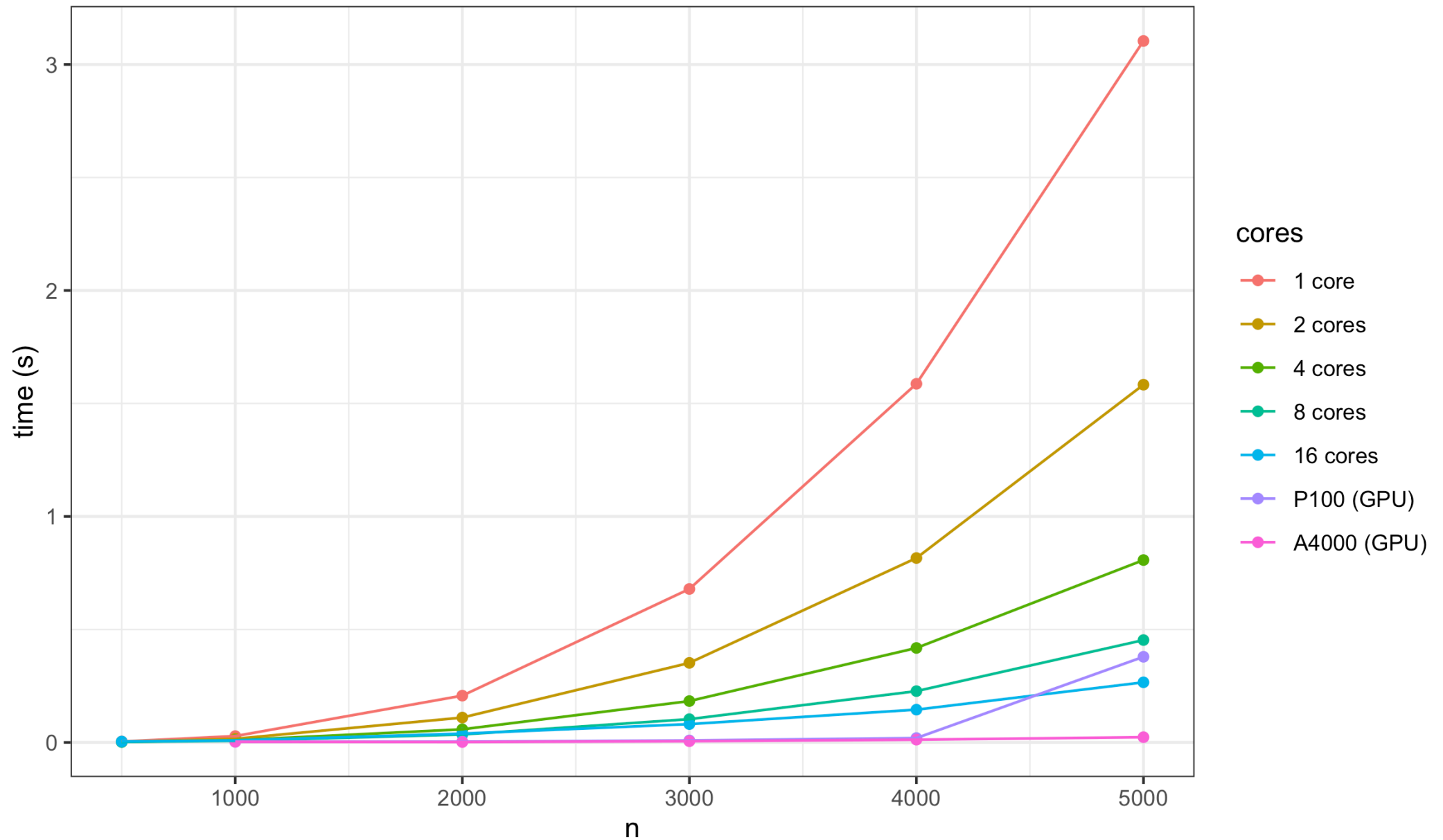
- ATLAS - Automatically Tuned Linear Algebra Software
- OpenBLAS - fork of GotoBLAS from TACC at UTexas
- Intel MKL - Math Kernel Library, part of Intel's commercial compiler tools
- cuBLAS / Magma - GPU libraries from Nvidia and UTK respectively
- Accelerate / vecLib - Apple's framework for GPU and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

OpenBLAS Matrix Multiply Performance

```
1 x=matrix(runif(5000^2),ncol=5000)
2
3 sizes = c(100,500,1000,2000,3000,4000,5000)
4 cores = c(1,2,4,8,16)
5
6 sapply(
7   cores,
8   function(n_cores) {
9     flexiblas::flexiblas_set_num_threads(n_cores)
10    sapply(
11      sizes,
12      function(s) {
13        y = x[1:s,1:s]
14        system.time(y %*% y)[3]
15      }
16    )
17  }
18 )
```

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



Matrix Multiply of (n x n) matrices

