

# Profiling & Benchmarking

# profvis demo

```
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))

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

# 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$x > 0.5, ]      150.8µs  156µs    5943.  251.49KB  14.9
## 2 d[which(d$x > 0.5), ] 211.1µs 216.61µs  4556.  267.34KB  22.9
## 3 subset(d, x > 0.5)  279.11µs 306.01µs  3241.  285.22KB  17.0
## 4 filter(d, x > 0.5)   1.82ms   1.94ms    515.   1.54MB   8.37
```

# 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`
  - `mcpParallel` & `mccollect`

# detectCores

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

```
detectCores()
```

```
## [1] 16
```

# pvec

## Parallelization of a vectorized function call

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

```
##   user  system elapsed  
## 0.214   0.029   0.243
```

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

```
##   user  system elapsed  
## 0.442   0.185   0.631
```

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

```
##   user  system elapsed  
## 0.532   0.389   0.372
```

# pvec - bench::system\_time

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

```
## process    real
```

```
##   180ms   180ms
```

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

```
## process    real
```

```
##   935ms   980ms
```

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

```
## process    real
```

```
##   1.01s   1.05s
```

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

```
## process    real
```

```
##   1.93ms 500.09ms
```

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

```
## user system elapsed
```

```
## 0.001 0.000 0.500
```



```
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, .)
```

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

res

```
##           10^6  10^7  10^8
## 1 cores  0.011 0.133 1.662
## 4 cores  0.059 0.360 4.664
## 8 cores  0.072 0.372 3.971
## 16 cores 0.098 0.432 3.979
```

# mclapply

## Parallelized version of lapply

```
system.time(rnorm(1e6))  
  
##   user  system elapsed  
## 0.101   0.007   0.107  
  
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 2)))  
  
##   user  system elapsed  
## 0.148   0.136   0.106  
  
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 4)))  
  
##   user  system elapsed  
## 0.242   0.061   0.052
```

```
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 4)))  
## user system elapsed  
## 0.097 0.047 0.079  
  
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 8)))  
## user system elapsed  
## 0.193 0.076 0.040  
  
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 10)))  
## user system elapsed  
## 0.162 0.083 0.041  
  
system.time(unlist(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 12)))  
## user system elapsed  
## 0.098 0.065 0.037
```

# mcpaerelle

Asynchronously evaluation of an R expression in a separate process

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

```
str(m)
```

```
## List of 2  
## $ pid: int 10870  
## $ fd : int [1:2] 4 7  
## - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```
str(n)
```

```
## List of 2  
## $ pid: int 10871  
## $ fd : int [1:2] 5 9  
## - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

# mccollect

Checks `mcpParallel` objects for completion

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

```
## List of 3
```

```
## $ 10870: num [1:1000000] -0.529 0.177 -0.494 1.46 -0.883 ...
```

```
## $ 10871: num [1:1000000] 0.21 0.196 0.467 0.657 0.776 ...
```

```
## $ 10872: num [1:1000000] 0.688 0.22 1.887 0.284 1.492 ...
```

# 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] 16  
  
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.

```
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

```
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
```

```
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.

```
registerDoMC(4)
```

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

```
##   user  system elapsed  
## 0.325  0.060  0.460
```

```
registerDoMC(8)
```

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

```
##   user  system elapsed  
## 0.679  0.096  0.483
```

```
registerDoMC(12)
```

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

```
##   user  system elapsed  
## 0.730  0.145  0.487
```



# 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.

```
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
)
```

```
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()
```

# 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

# 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).

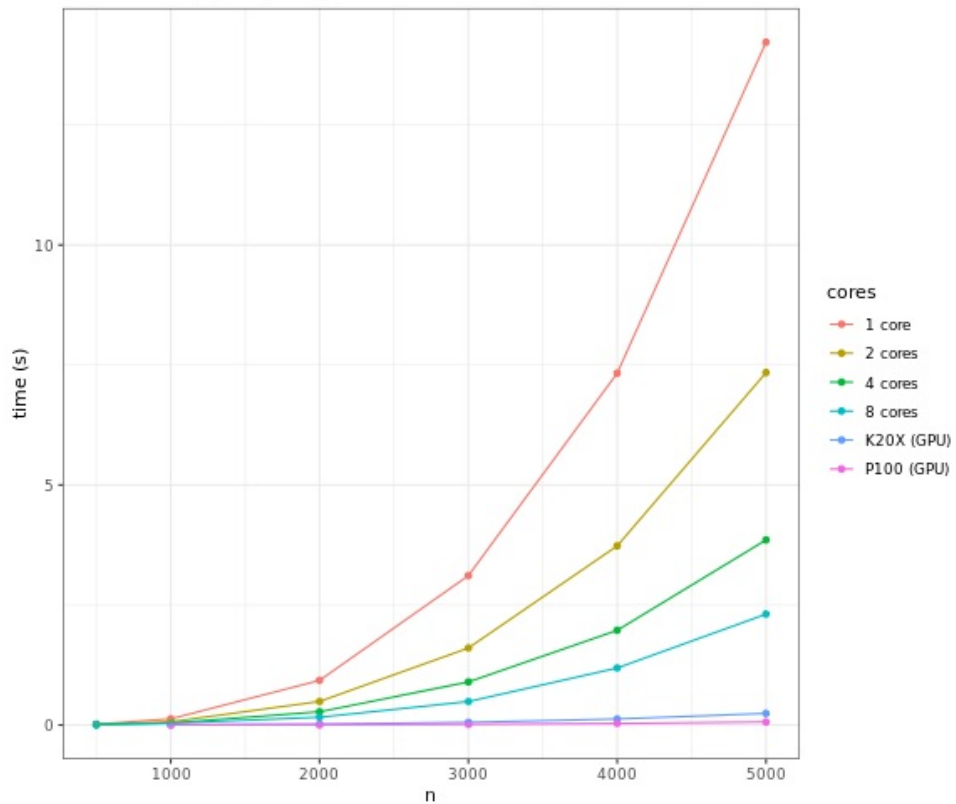
Multithreaded alternatives:

- 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

# OpenBLAS Matrix Multiply (DGEMM) Performance

n	1 core	2 cores	4 cores	8 cores
100	0.001	0.001	0.000	0.000
500	0.018	0.011	0.008	0.008
1000	0.128	0.068	0.041	0.036
2000	0.930	0.491	0.276	0.162
3000	3.112	1.604	0.897	0.489
4000	7.330	3.732	1.973	1.188
5000	14.223	7.341	3.856	2.310

Matrix Multiply of (n x n) matrices - double precision



Matrix Multiply of (n x n) matrices - double precision

