

Parallelization

Statistical Computing & Programming

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- Multicore Data Science with R and Python
- Beyond Single-Core R [slides](#) by Jonathan Dursi
- Getting started with `doMC` and `foreach` [vignette](#) by Steve Weston

Timing code

Benchmarking with package bench

```
library(bench)

x <- runif(n = 1000000)
b <- bench::mark(
  sqrt(x),
  x ^ 0.5,
  x ^ (1 / 2),
  exp(log(x) / 2),
  time_unit      = 's'
)
b

#> # A tibble: 4 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>    <dbl>    <dbl>     <dbl> <bch:byt>    <dbl>
#> 1 sqrt(x)      0.00219  0.00268     334.   7.63MB     110.
#> 2 x^0.5        0.0189   0.0192     51.5   7.63MB     16.3
#> 3 x^(1/2)      0.0189   0.0196     51.0   7.63MB     17.0
#> 4 exp(log(x)/2) 0.0135  0.0141     68.8   7.63MB     11.0
```

If one of 'ns', 'us', 'ms', 's', 'm', 'h', 'd', 'w' the time units are instead expressed as nanoseconds, microseconds, milliseconds, seconds, hours, minutes, days or weeks respectively.

Relative performance

```
class(b)

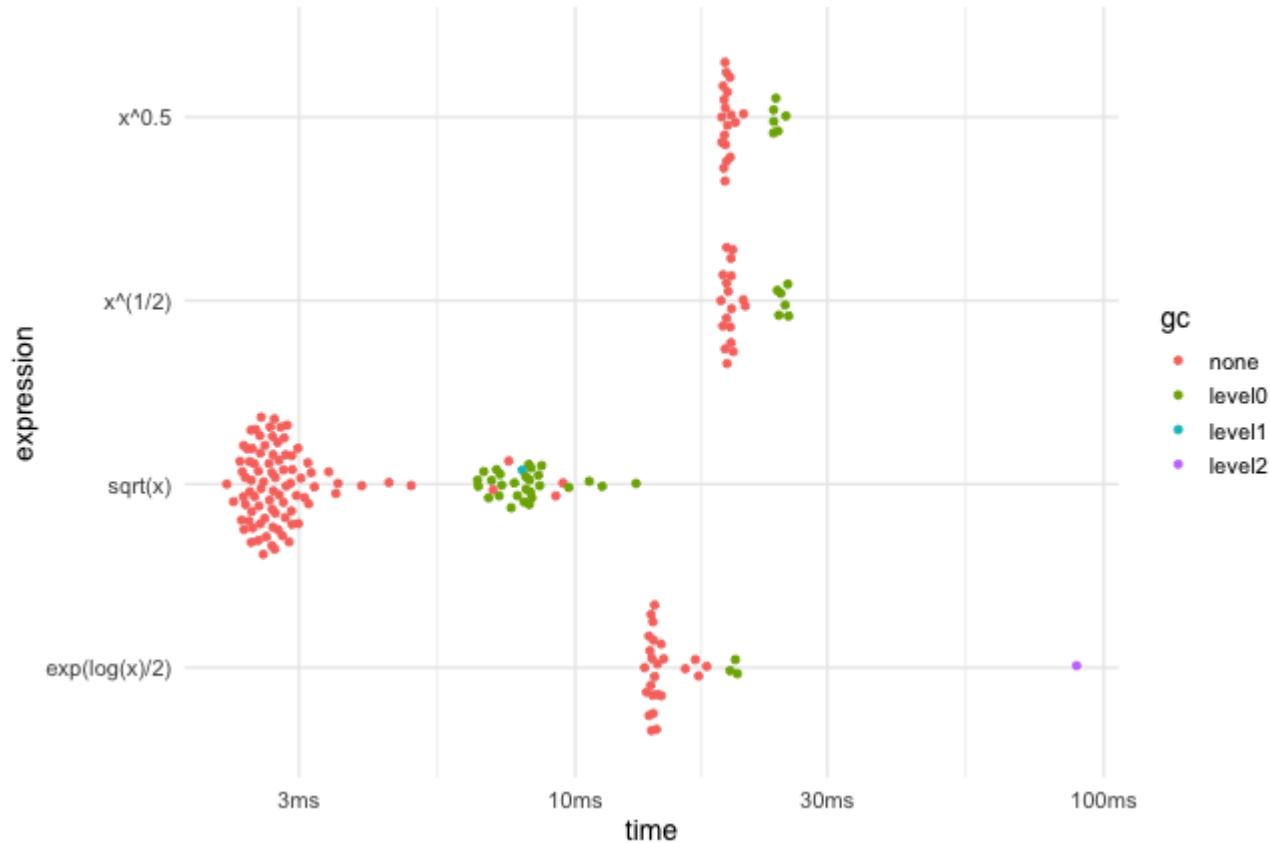
#> [1] "bench_mark" "tbl_df"       "tbl"          "data.frame"

summary(b, relative = TRUE)

#> # A tibble: 4 x 6
#>   expression      min  median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>    <dbl>    <dbl>     <dbl>      <dbl>    <dbl>
#> 1 sqrt(x)        1       1       6.54      1       9.98
#> 2 x^0.5         8.65    7.18     1.01      1       1.48
#> 3 x^(1/2)       8.62    7.30     1         1       1.54
#> 4 exp(log(x)/2) 6.18    5.25     1.35      1       1
```

Visualize the performance

```
plot(b) + theme_minimal(base_size = 14)
```



CPU and real time

```
system.time({
  x <- c()
  for (i in 1:100000) {
    x <- c(x, rnorm(1) + 5)
  }
})
```

```
#>      user  system elapsed
#>  15.990   8.803  25.024
```

```
system.time({
  x <- numeric(length = 100000)
  for (i in 1:100000) {
    x[i] <- rnorm(1) + 5
  }
})
```

```
#>      user  system elapsed
#>  0.195   0.051   0.248
```

```
system.time({
  rnorm(100000) + 5
})
```

```
#>      user  system elapsed
#>  0.006   0.000   0.007
```

```
x <- data.frame(matrix(rnorm(100000), nrow = 1))
```

```
bench_time({  
  types <- numeric(dim(x)[2])  
  for (i in seq_along(x)) {  
    types[i] <- typeof(x[i])  
  }  
})
```

```
#> process      real  
#>   6.75s     6.79s
```

```
bench_time({  
  sapply(x, typeof)  
})
```

```
#> process      real  
#>   73.7ms    74.2ms
```

```
bench_time({  
  purrr::map_chr(x, typeof)  
})
```

```
#> process      real  
#>   507ms    511ms
```

Exercises

1. Benchmark `which("q" == chars)[1]` and `match("q", chars)`, where

```
chars <- sample(c(letters, 0:9), size = 1000,  
                 replace = TRUE)
```

What do these expression do?

2. Benchmark the last two expression.

```
X <- matrix(rnorm(1000 * 1000), nrow = 1000, ncol = 1000)  
  
sum(diag(X %*% t(X)))  
sum(X ^ 2)
```

What do these expression do?

3. Investigate

```
bench_time(Sys.sleep(3))  
  
bench_time(read.csv(str_c("http://www2.stat.duke.edu/~sms185/",  
                      "data/bike/cbs_2013.csv")))
```

Parallelization

Code bounds

Your R [substitute a language] computations are typically bounded by some combination of the following four factors.

1. CPUs
2. Memory
3. Inputs / Outputs
4. Network

Today we'll focus on how our computations (in some instances) can be less affected by the first bound.

Terminology

- **CPU:** central processing unit, primary component of a computer that processes instructions
- **Core:** an individual processor within a CPU, more cores will improve performance and efficiency
 - You can get a Duke VM with 2 cores
 - Your laptop probably has 2, 4, or 8 cores
 - DSS R cluster has 16 cores
 - Duke's computing cluster (DCC) has 15,667 cores
- **User CPU time:** the CPU time spent by the current process, in our case, the R session
- **System CPU time:** the CPU time spent by the OS on behalf of the current running process

Run in serial or parallel

Suppose I have n tasks, t_1, t_2, \dots, t_n , that I want to run.

To **run in serial** implies that first task t_1 is run and we wait for it to complete. Next, task t_2 is run. Upon its completion the next task is run, and so on, until task t_n is complete. If each task takes s seconds to complete, then my theoretical run time is sn .

Assume I have n cores. To **run in parallel** means I can divide my n tasks among the n cores. For instance, task t_1 runs on core 1, task t_2 runs on core 2, etc. Again, if each task takes s seconds to complete and I have n cores, then my theoretical run time is s seconds - this is never the case. *Here we assume all n tasks are independent.*

Ways to parallelize

Forking: a copy of the current R session is moved to new cores.

- Not available on Windows
- Less overhead and easy to implement

Sockets: a new R session is launched on each core.

- Available on all systems
- Each process on each core is unique

Package parallel

This package builds on packages `snow` and `multicore`. It can handle much larger chunks of computation in parallel.

```
library(parallel)
```

Core functions:

- `detectCores()`
- `pvec()`, based on forking
- `mclapply()`, based on forking
- `mcparallel()`, `mccollect()`, based on forking

Follow along on our DSS R server or your own machine.

How many cores do I have?

On my MacBook Pro

```
detectCores()
```

```
#> [1] 8
```

On the DSS servers

```
detectCores()
```

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

pvec ()

Using forking, `pvec()` parallelizes the execution of a function on vector elements by splitting the vector and submitting each part to one core.

```
system.time(rnorm(1e7) ^ 4)
```

```
#>      user  system elapsed
#> 0.839    0.022   0.868
```

```
system.time(pvec(v = rnorm(1e7), FUN = `^`, 4, mc.cores = 1))
```

```
#>      user  system elapsed
#> 0.832    0.021   0.857
```

```
system.time(pvec(v = rnorm(1e7), FUN = `^`, 4, mc.cores = 2))
```

```
#>      user  system elapsed
#> 1.510    0.586   1.587
```

```
system.time(pvec(v = rnorm(1e7), FUN = `^`, 4, mc.cores = 4))
```

```
#>    user  system elapsed
#> 1.049   0.263   0.957
```

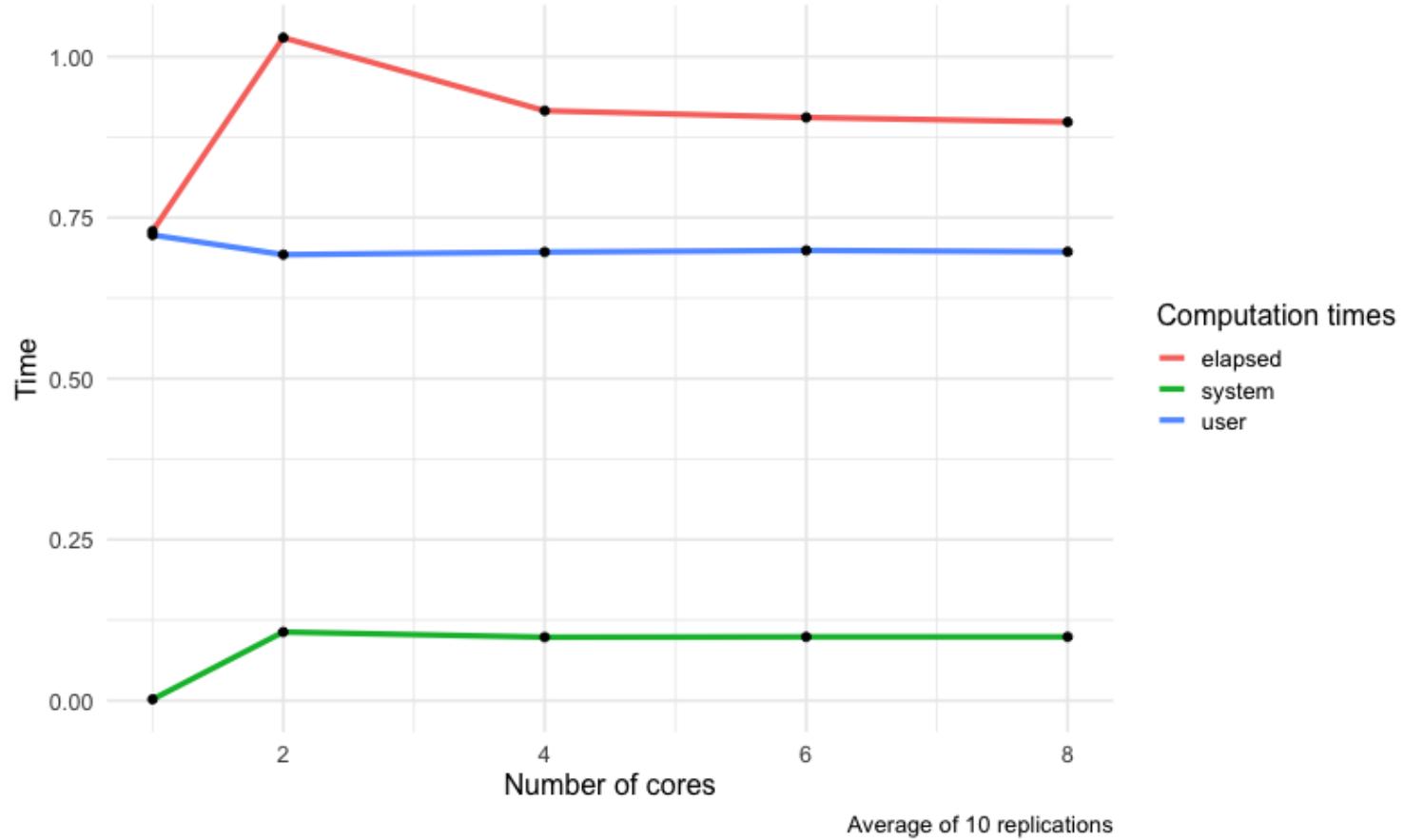
```
system.time(pvec(v = rnorm(1e7), FUN = `^`, 4, mc.cores = 6))
```

```
#>    user  system elapsed
#> 1.116   0.259   0.906
```

```
system.time(pvec(v = rnorm(1e7), FUN = `^`, 4, mc.cores = 8))
```

```
#>    user  system elapsed
#> 1.167   0.311   0.923
```

```
pvec(v = rnorm(1e7), FUN = '^', 4, mc.cores = *)
```



Don't underestimate the overhead cost!

mclapply()

Using forking, mclapply() is a parallelized version of lapply(). Recall that lapply() returns a list, similar to map().

```
system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 1))  
#>   user  system elapsed  
#> 0.058    0.000   0.060  
  
system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 2))  
#>   user  system elapsed  
#> 0.148    0.136   0.106  
  
system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 4))  
#>   user  system elapsed  
#> 0.242    0.061   0.052
```

```
system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 6))

#>   user  system elapsed
#> 0.113  0.043  0.043

system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 8))

#>   user  system elapsed
#> 0.193  0.076  0.040

system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 10))

#>   user  system elapsed
#> 0.162  0.083  0.041

system.time(mclapply(1:10, function(x) rnorm(1e5), mc.cores = 12))

#>   user  system elapsed
#> 0.098  0.065  0.037
```

Another example

```
delayed_rpois <- function(n) {  
  Sys.sleep(1)  
  rpois(n, lambda = 3)  
}
```

```
bench_time(mclapply(1:8, delayed_rpois, mc.cores = 1))
```

```
#> process     real  
#>  3.27ms    8.02s
```

```
bench_time(mclapply(1:8, delayed_rpois, mc.cores = 4))
```

```
#> process     real  
#>  5.89ms    2.01s
```

```
bench_time(mclapply(1:8, delayed_rpois, mc.cores = 8))
```

```
#> process     real  
#> 18.89ms   1.02s
```

```
# I don't have 800 cores  
bench_time(mclapply(1:8, delayed_rpois, mc.cores = 800))
```

```
#> process     real  
#> 11.33ms   1.01s
```

mcpallel()

Using forking, evaluate an R expression asynchronously in a separate process. Function `mcpallel()` starts a parallel R process which evaluates the given expression.

```
x <- mcpallel({  
  cat("I'm evaluated asynchronously")  
  mean(c(6, 1, 4, -2, 4, 3))  
})
```

```
x
```

```
#> $pid  
#> [1] 93127  
#>  
#> $fd  
#> [1] 4 7  
#>  
#> attr(,"class")  
#> [1] "parallelJob" "childProcess" "process"
```

```
Sys.getpid()
```

```
#> [1] 93101
```

mccollect()

Collect your result(s) with `mccollect()`

```
mccollect(x)
```

```
#> $`93127`  
#> [1] 2.666667
```

List example

```
x <- list()

x$pois <- mcparallel({
  Sys.sleep(1)
  rpois(10, 2)
} )

x$norm <- mcparallel({
  Sys.sleep(2)
  rnorm(10)
} )

x$beta <- mcparallel({
  Sys.sleep(3)
  rbeta(10, 1, 1)
} )

result <- mccollect(x)
str(result)
```

```
#> List of 3
#> $ 93128: int [1:10] 1 2 1 1 2 5 1 3 0 1
#> $ 93129: num [1:10] 0.4466 0.4024 0.8942 0.4715 -0.0856 ...
#> $ 93130: num [1:10] 0.846 0.666 0.388 0.16 0.336 ...
```

```
bench_time({  
  x <- list()  
  
  x$pois <- mcpallel({  
    Sys.sleep(1)  
    rpois(10, 2)  
  })  
  
  x$norm <- mcpallel({  
    Sys.sleep(2)  
    rnorm(10)  
  })  
  
  x$beta <- mcpallel({  
    Sys.sleep(3)  
    rbeta(10, 1, 1)  
  })  
  
  result <- mccollect(x)  
})
```

```
#> process     real  
#>   4.06ms   3.01s
```

How many cores are being used?

More mccollect()

To check some of your results early set `wait = FALSE` and a timeout time in seconds.

```
p <- mcparallel({  
  Sys.sleep(1)  
  x <- rnorm(1000)  
  c(mean(x), var(x))  
})  
  
mccollect(p)  
  
#> $`93150`  
#> [1] -0.01603019 0.94998873
```

However, if you are impatient, you may get a NULL value.

```
q <- mcparallel({  
  Sys.sleep(1)  
  x <- rnorm(1000)  
  c(mean(x), var(x))  
})  
  
mccollect(q, wait = FALSE)
```

```
#> NULL
```

```
mccollect(q)  
  
#> $`93151`  
#> [1] 0.01416182 0.90857523
```

Exercises

1. Do you notice anything strange with objects `result2` and `result4`? What is going on?

```
result2 <- mclapply(1:12, FUN = function(x) rnorm(1),  
                     mc.cores = 2, mc.set.seed = FALSE) %>% unlist()  
result2
```

```
#> [1] 1.3142269 1.3142269 -0.8261026 -0.8261026 -0.6428599 -0.6428599  
#> [7] -1.7795643 -1.7795643 -1.3317912 -1.3317912 1.3516840 1.3516840
```

```
result4 <- mclapply(1:12, FUN = function(x) rnorm(1),  
                     mc.cores = 4, mc.set.seed = FALSE) %>% unlist()  
result4
```

```
#> [1] 1.3142269 1.3142269 1.3142269 1.3142269 -0.8261026 -0.8261026  
#> [7] -0.8261026 -0.8261026 -0.6428599 -0.6428599 -0.6428599 -0.6428599
```

2. Parallelize the evaluation of the four expressions below.

```
mtcars %>%
  count(cyl)

mtcars %>%
  lm(mpg ~ wt + hp + factor(cyl), data = .)

map_chr(mtcars, typeof)

mtcars %>%
  select(mpg, disp:qsec) %>%
  map_df(summary)
```

3. Suppose you only have two cores. Use `mclapply()` to execute the below function in parallel so the total run-time is six seconds.

```
sleep_r <- function(x) {  
  Sys.sleep(x)  
  runif(1)  
}  
  
x <- c(3, 3, 6)
```

References

1. Beyond Single-Core R. <https://ljdursi.github.io/beyond-single-core-R/#/>.
2. Jones, M. (2021). Quick Intro to Parallel Computing in R. <https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html>.
3. Parallel (2021). <https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf>.