Parallelization

Programming for Statistical Science

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)       0.00213 0.00244     347.     7.63MB    83.6
#> 2 x^0.5         0.0166  0.0185       54.1    7.63MB     9.84
#> 3 x^(1/2)       0.0173  0.0181       54.8    7.63MB     6.85
#> 4 exp(log(x)/2) 0.0126  0.0137       73.2    7.63MB    11.8
```

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)
```

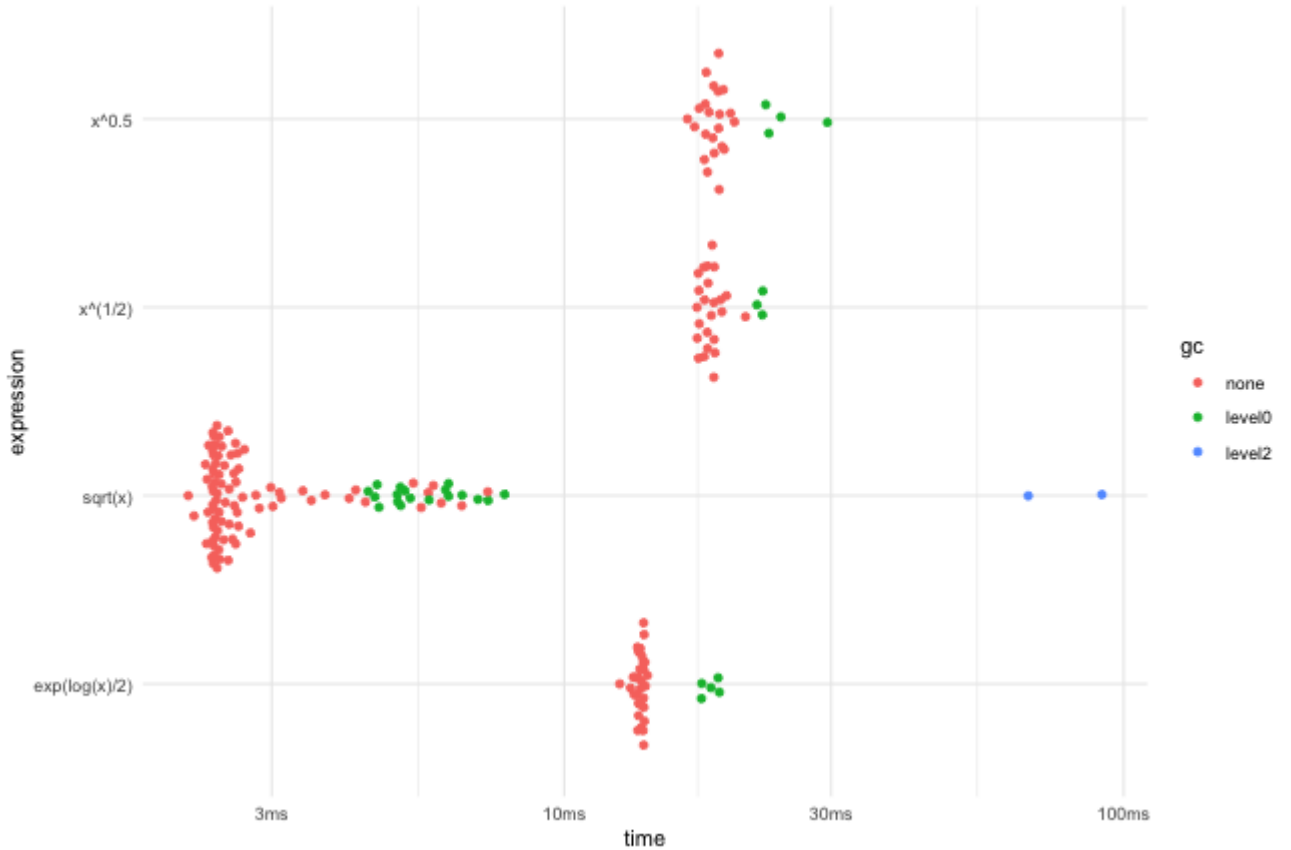
```
#> [1] "bench_mark" "tbl_df"     "tbl"        "data.frame"
```

```
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         6.41         1    12.2
#> 2 x^0.5           7.81   7.59      1            1     1.44
#> 3 x^(1/2)         8.12   7.41      1.01         1     1
#> 4 exp(log(x)/2)   5.91   5.63      1.35         1     1.72
```

Visualize the performance

```
plot(b) + theme_minimal()
```



CPU and real time

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

```
#>    user  system elapsed
#>  19.039   9.692  28.824
```

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

```
#>    user  system elapsed
#>   0.181   0.032   0.214
```

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

```
#>    user  system elapsed
#>   0.007   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
#>   6.91s   6.96s
```

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

```
#> process    real
#>  97.2ms  97.3ms
```

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

```
#> process    real
#>   474ms   475ms
```

Exercises

1. Compare `which("q" == sample_letters)[1]` and `match("q", sample_letters)`, where

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

   What do these expression do?

2. 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, ..., 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

1. Sockets

   A new version of R is launched on each core.
   - Available on all systems
   - Each process on each core is unique

2. Forking

   A copy of the current R session is moved to new cores.
   - Not available on Windows
   - Less overhead and easy to implement

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 cluster.

How many cores do I have?

On my MacBook Pro

```
detectCores()
```

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

On pawn, rook, knight

```
detectCores()
```

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

pvec()

Using forking, `pvec()` parellelizes 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
#>   0.825   0.021   0.846
```

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

```
#>    user  system elapsed
#>   0.831   0.017   0.848
```

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

```
#>    user  system elapsed
#>   1.527   0.556   1.581
```

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

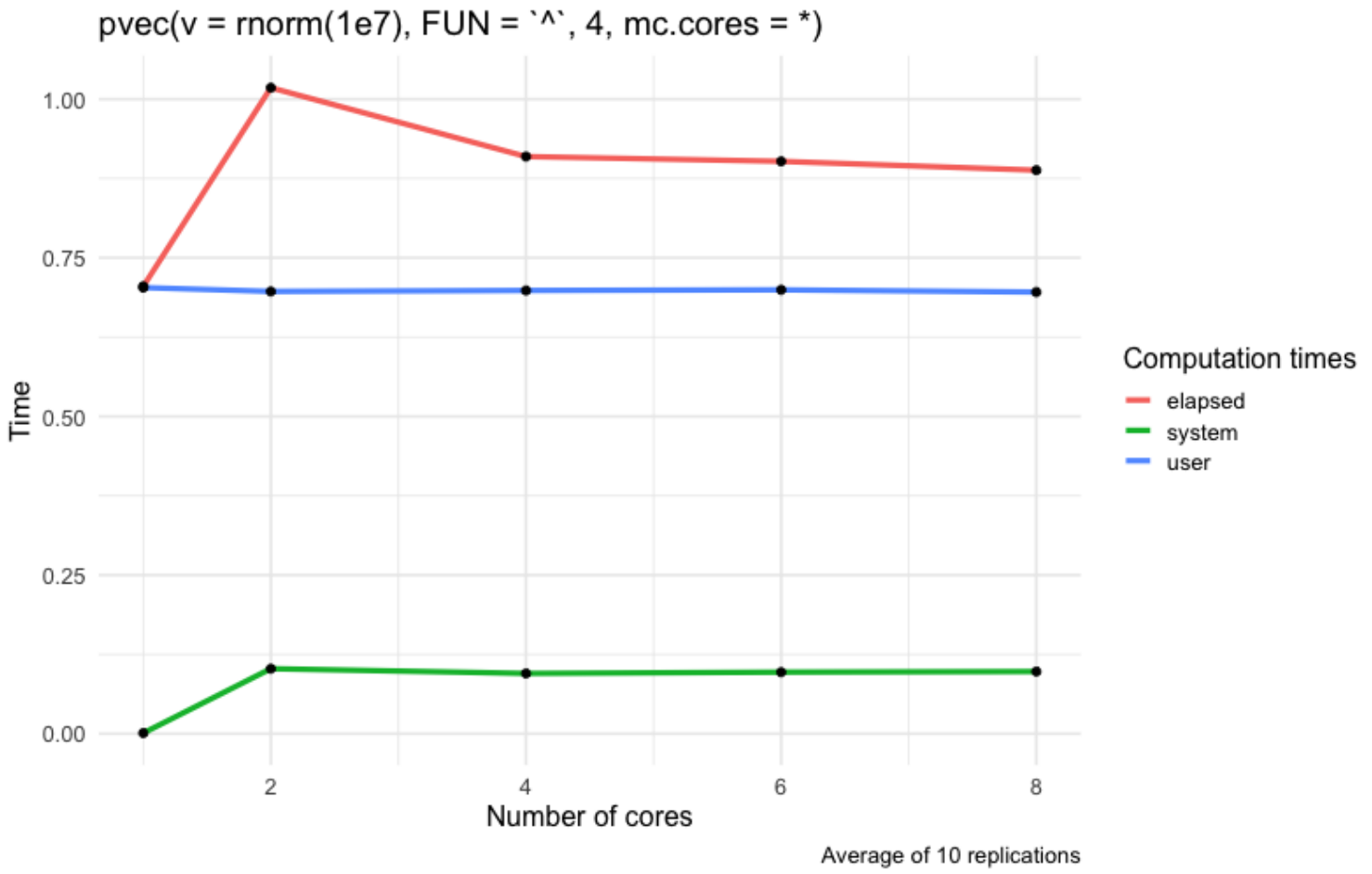
```
#>    user  system elapsed
#>   1.115   0.296   0.994
```

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

```
#>    user  system elapsed
#>   1.116   0.236   0.905
```

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

```
#>    user  system elapsed
#>   1.181   0.291   0.894
```



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

#>    user  system elapsed
#>   0.058   0.000   0.060

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
```

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

#>    user  system elapsed
#>   0.113   0.043   0.043

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
```

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
#>  5.57ms   8.03s
```

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

```
#> process    real
#>  20.8ms   2.02s
```

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

```
#> process    real
#> 13.29ms   1.01s
```

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

```
#> process    real
#> 10.62ms   1.01s
```

`mcparallel()` & `mccollect()`

Using forking, evaluate an R expression asynchronously in a separate process.

```
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
#>  $ 43765: int [1:10] 2 4 2 2 2 2 3 2 2 4
#>  $ 43766: num [1:10] -1.151 -1.931 -0.182 -1.222 -1.023 ...
#>  $ 43767: num [1:10] 0.999 0.539 0.241 0.435 0.101 ...
```

```
bench_time({
  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)
})
```

```
#> process    real
#>  3.88ms   3.01s
```

A closer look at `mcparallel()` & `mccollect()`

```
str(x)
```

```
#> List of 3
#>  $ pois:List of 2
#>   ..$ pid: int 43776
#>   ..$ fd : int [1:2] 4 7
#>   ..- attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
#>  $ norm:List of 2
#>   ..$ pid: int 43777
#>   ..$ fd : int [1:2] 5 9
#>   ..- attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
#>  $ beta:List of 2
#>   ..$ pid: int 43778
#>   ..$ fd : int [1:2] 6 11
#>   ..- attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

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

```
p <-  mcparallel({
  Sys.sleep(1)
  mean(rnorm(100))
  })

mccollect(p, wait = FALSE, timeout = 2)
```

```
#> $`43780`
#> [1] 0.1254564
```

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

```
q <-  mcparallel({
  Sys.sleep(1)
  mean(rnorm(100))
  })

mccollect(q, wait = FALSE)
```

```
#> NULL
```

```
mccollect(q)
```

```
#> $`43789`
#> [1] 0.06071482
```

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]  1.4194792  1.4194792 -1.6042415 -1.6042415 -0.8597708 -0.8597708
#>  [7]  0.9880630  0.9880630 -0.6827594 -0.6827594 -0.8258170 -0.8258170
```

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

```
#>  [1]  1.4194792  1.4194792  1.4194792  1.4194792 -1.6042415 -1.6042415
#>  [7] -1.6042415 -1.6042415 -0.8597708 -0.8597708 -0.8597708 -0.8597708
```

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)
```

Sockets

Using sockets to parallelize

The basic recipe is as follows:

```
library(parallel)

detectCores()
c1 <- makeCluster()
result <- clusterApply(cl = c1, ...)
stopCluster(c1)
```

Here you are spawning new R sessions. Data, packages, functions, etc. need to be shipped to the workers.

We'll go into more details on using sockets next lecture.

References

1. Beyond Single-Core R. https://ljdursi.github.io/beyond-single-core-R/#/.

2. Jones, M. (2020). Quick Intro to Parallel Computing in R. https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html.

3. Parallel (2020). https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf.