TRUMAN STATE UNIVERSITY

Professional Program in
DATA SCIENCE

Parallel Computing in R

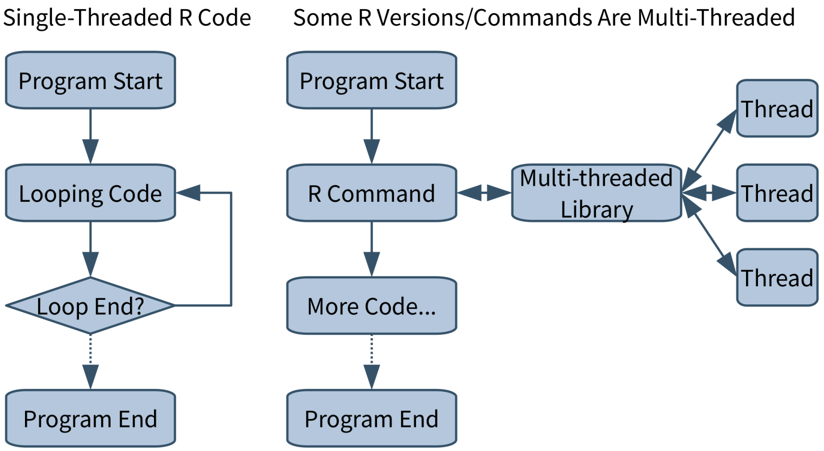
DISTINCT BY DESIGN

Introduction to Parallelization

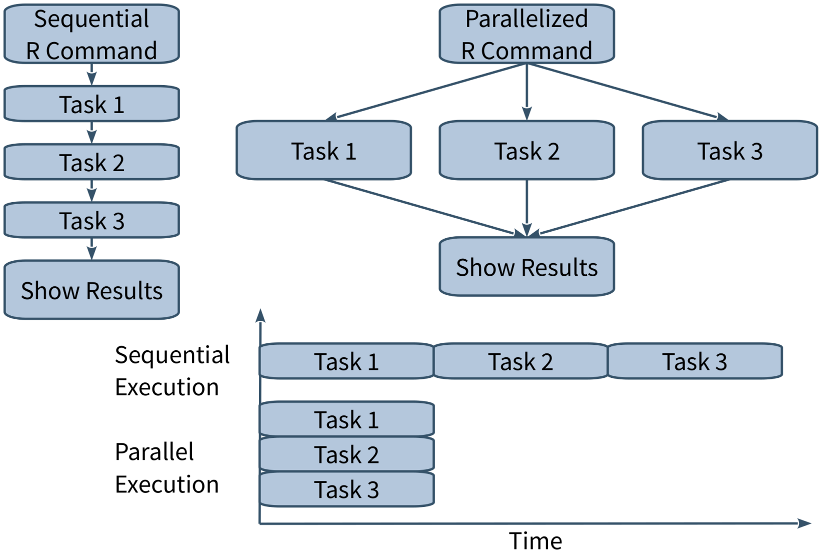
Goals for this Lecture

- Review pros and cons of parallel/distributed computing
- Introduce parallel computing methods in R
- Methods that parallelize general loops
    - The `parallel` package and `mclapply` commands.
    - The `foreach` and `doParallel` packages.
- Methods that are `dplyr`-aware
    - The `multidplyr` package
    - Local Spark instances with `sparklyr`
- Types of problems that are amenable to each approach
- Effects of adding cores/clusters

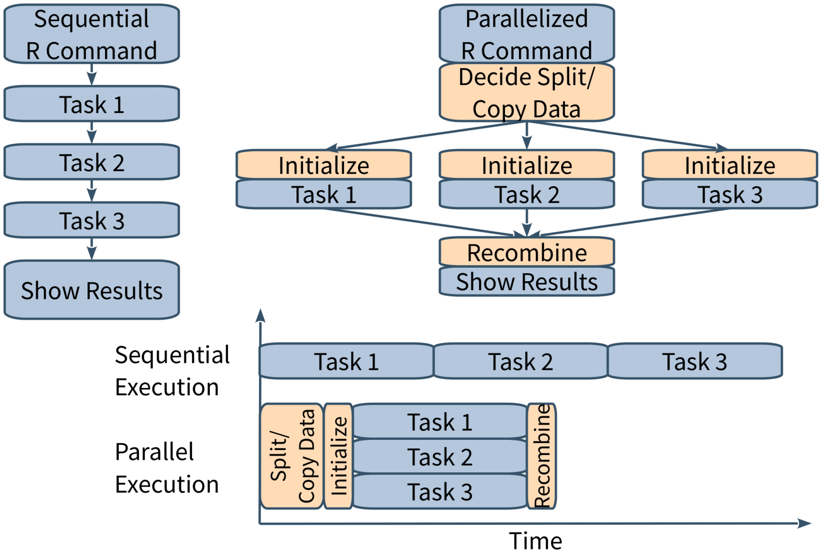
R is single-threaded by design, but there are exceptions.



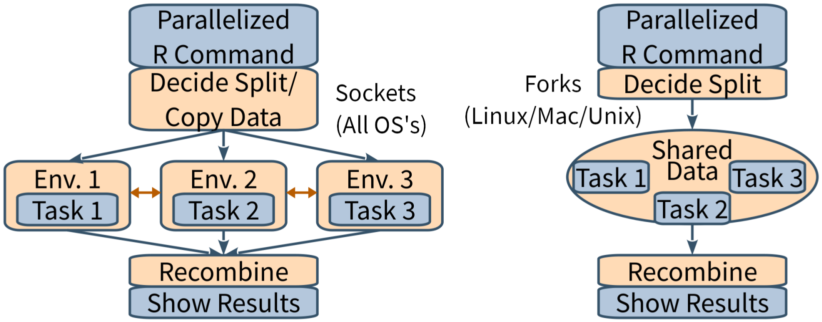
In a perfect world, parallelization would be 100% efficient.



In reality, there is always overhead in splitting a task. How much depends on the nature of the task.



Efficiency is also affected by whether or not processes share memory, and by how they must communicate.



- Separate environments are more costly in time and resources.
- Communication between tasks adds complexity.

Task assignment can be pre-scheduled or dynamic.

- Pre-scheduling breaks the tasks into chunks equal to the number of cores/threads/cluster nodes at the beginning.
    - Good for short, similar multitudinous tasks.
    - Good when task length can be easily estimated for balancing.
- Dynamic assignment sends the next task to the next available core/thread/node.
    - Good when task time is highly variable.
    - Good for smaller numbers of tasks.
    - More overhead?

Simulation: Parallelizing Loops

Hypothetical Example: Testing a weight-loss program.

- You want to test a new weight loss program over one month.
- *If the program works*, it would be reasonable for subjects to lose five pounds, on average.
- Assume two groups with a real difference in means:
  - Control group. Mean weight loss = 0, std. dev. = 3 lbs.
  - Treatment group. Mean weight loss = 5, std. dev. = 3 lbs.
- Will two groups of size *n* be able to detect that difference?

```
diff.test <- function(n=5, a=0.05){
  t.test(x=rnorm(n, mean=0, sd=3), y=rnorm(n, mean=5, sd=3))$p.value < a
}
diff.test(n=5, a=0.05)
```

```
## [1] FALSE
```

Statistical *power* is the probability that a hypothesis test will detect a true difference between groups.

- Power *can* be calculated theoretically in some cases.
- Simulation can be used to calculate power even when theory is difficult.
  - Assume a certain probability distribution for each group.
  - Simulate a random sample from each group's distribution.
  - Do the test, and record whether the test detects a difference (rejects H_0). [This is the `test.diff` function.]
  - Repeat.
  - Calculate the percentage of runs that detect a difference.

Without parallelization, we could implement the simulation in *many* ways!

```r
# Replicate repeats expressions
# that don't need inputs.
test.vec <- replicate(10000, diff.test())
mean(test.vec)
```

```
## [1] 0.6064
```

```r
# lapply plugs each value of n.vec into
# the function and returns a list.
n.vec <- rep(5, 10000)
test.vec <- lapply(n.vec, diff.test)
mean(unlist(test.vec))
```

```
## [1] 0.61
```

```r
# A standard `for` loop to do the same
test.vec <- numeric(10000)
for(i in 1:10000){
  test.vec[i] <- diff.test()
}
mean(test.vec)
```

```
## [1] 0.608
```

```r
# `foreach` gives a new syntax, which acts
# like a `for` loop, but combines output.
library(foreach)
test.vec <-
  foreach(n=n.vec, .combine=c) %do%
    diff.test(n)
mean(test.vec)
```

```
## [1] 0.6237
```

```r
# The `map` family from `purrr` are similar
# to `lapply` for predefined functions.
test.vec <- map_dbl(n.vec, diff.test)
# but with a nice function construction
# syntax.
test.vec <- map_dbl(n.vec,
  ~t.test(x=rnorm(.x, mean=0, sd=3),
          y=rnorm(.x, mean=5, sd=3))$p.value
mean(test.vec)
```

```
## [1] 0.6091
```

the parallel package provides mclapply which uses the standard lapply syntax. (linux/unix/mac)

Single-thread computation

```r
# want 10000 repetitions, and
# lapply needs an input.
n.vec <- rep(5, 10000)
system.time({
  test.vec <-
    lapply(n.vec, diff.test)
})
```

```
##    user  system elapsed
##   1.324   0.000   1.324
```

```r
test.power <-
  mean(unlist(test.vec))
test.power
```

```
## [1] 0.6111
```

Multi-thread computation

```r
# note: mclapply requires forks.
# needs linux/mac (not windows).
library(parallel)
n.cores <- detectCores()-1
system.time({
  test.vec <- mclapply(n.vec,
                       diff.test,
                       mc.cores=n.cores)
})
```

```
##    user  system elapsed
##   2.102   0.341   0.519
```

```r
test.power <-
  mean(unlist(test.vec))
test.power
```

```
## [1] 0.6078
```

The parallel package also provides parLapply which is used with pre-defined clusters (all platforms).

Timing Excluding Initialization

```
cl <- makeCluster(n.cores)
system.time({
  test.vec <- parLapply(cl, n.vec,
                        diff.test)
})
```

```
##    user  system elapsed
##   0.001   0.004   0.402
```

```
# Make sure to stop your cluster!
stopCluster(cl)
test.power <-
  mean(unlist(test.vec))
test.power
```

```
## [1] 0.6163
```

Initialization adds Overhead

```
system.time({
cl <- makeCluster(n.cores)
  test.vec <- parLapply(cl, n.vec,
                        diff.test)
# Make sure to stop your cluster!
stopCluster(cl)
})
```

```
##    user  system elapsed
##   0.010   0.000   0.983
```

```
test.power <-
  mean(unlist(test.vec))
test.power
```

```
## [1] 0.6131
```

The doParallel package provides an parallel backend to the foreach package with platform-independent syntax.

Single-Thread Computation

```r
library(foreach)
system.time({
  test.vec <- foreach(n=n.vec) %do%
                diff.test(n)
})
```

```
##    user  system elapsed
##   3.221   0.000   3.220
```

```r
test.power <-
  mean(unlist(test.vec))
test.power
```

```
## [1] 0.6137
```

Multi-Thread Computation

```r
library(doParallel)
registerDoParallel(n.cores)
system.time({
  test.vec <-
  foreach(n=n.vec, .combine=c) %dopar%
    diff.test(n)
})
```

```
##    user  system elapsed
##   3.973   0.412   1.598
```

```r
test.power <- mean(test.vec)
test.power
```

```
## [1] 0.6107
```

Note: `registerDoParallel(<number>)` selects the correct method for your system, and automatically cleans up clusters when done.

When cluster initialization is included, forks (Unix/Linux) appear faster than PSOCK clusters (all platforms).

Forks

```r
system.time({
cl <-
  makeCluster(n.cores, type="FORK")
registerDoParallel(cl)
test.vec <-
  foreach(n=n.vec) %dopar%
    diff.test(n)
})
```

```
##    user  system elapsed
##   2.519   0.276   2.820
```

```r
stopCluster(cl)
test.power <- mean(unlist(test.vec))
test.power
```

```
## [1] 0.6167
```

PSOCK Clusters

```r
system.time({
cl <-
  makeCluster(n.cores, type="PSOCK")
registerDoParallel(cl)
test.vec <-
  foreach(n=n.vec) %dopar%
    diff.test(n)
})
```

```
##    user  system elapsed
##   2.596   0.216   3.324
```

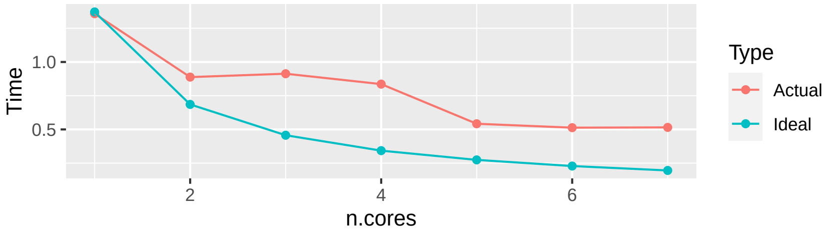
```r
stopCluster(cl)
test.power <- mean(unlist(test.vec))
test.power
```

```
## [1] 0.6152
```

Note: If clusters are defined explicitly, they must be stopped as well.

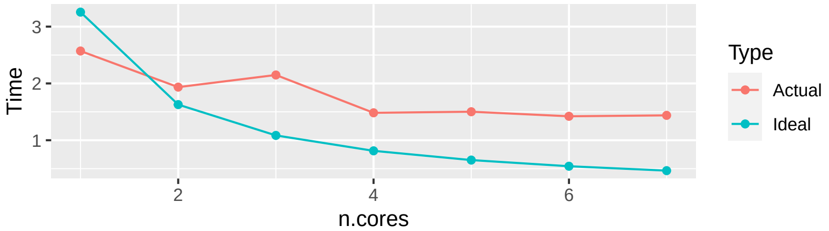
Because of additional overhead, completion time does not decrease much past a certain number of cores.

```r
# sapply loops through numbers of cores and creates a vector of results.
time.vec <- sapply(1:n.cores, function(cores){
  system.time({
    test.vec <- mclapply(n.vec, diff.test, mc.cores=cores)
  })["elapsed"]
})
data.frame(n.cores = 1:n.cores, Actual = time.vec,
  Ideal = system.time(lapply(n.vec, diff.test))["elapsed"]/(1:n.cores)) %>%
  pivot_longer(-n.cores, names_to="Type", values_to="Time") %>%
  ggplot(aes(x=n.cores, y=Time, color=Type)) + geom_point() + geom_line()
```



Because of additional overhead, completion time does not decrease much past a certain number of cores.

```r
time.vec <- sapply(1:n.cores, function(cores){
  registerDoParallel(cores)
  system.time({
    test.vec <- foreach(n=n.vec, .combine=c) %dopar% diff.test(n)
  })["elapsed"]
})
ideal.t <- system.time(foreach(n=n.vec) %do% diff.test(n))["elapsed"]
data.frame(n.cores = 1:n.cores, Actual = time.vec, Ideal=ideal.t/(1:n.cores)
  pivot_longer(-n.cores, names_to="Type", values_to="Time") %>%
  ggplot(aes(x=n.cores, y=Time, color=Type)) + geom_point() + geom_line()
```



Data Manipulation: Parallelizing `dplyr`

Example: Iris Averages

What are the average values for the parameters measured in the `iris` data set?

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

A Single-Core Solution

```
my.iris <- iris %>%
  rename(SL=Sepal.Length, SW=Sepal.Width,
         PL=Petal.Length, PW=Petal.Width)
my.iris %>% group_by(Species) %>% summarize(across(everything(), mean)
)
```

```
## # A tibble: 3 x 5
##   Species       SL    SW    PL    PW
##   <fct>      <dbl> <dbl> <dbl> <dbl>
## 1 setosa      5.01  3.43  1.46 0.246
## 2 versicolor  5.94  2.77  4.26 1.33
## 3 virginica   6.59  2.97  5.55 2.03
```

The `multidplyr` package creates a new `dplyr` backend that uses parallel processing.

Existing `dplyr` code can be used, with only a few additions:

- `cl <- new_cluster(<number>)` defines a cluster with `<number>` cores.
- Each cluster runs a new R instance, and must be initialized.
    - `cluster_library(cl, <packages>)` loads libraries from character vector `<packages>`.
    - `cluster_copy(cl, <names>)` copies variables from character vector `<names>`.
- `partition(cl)` assigns data frame groups to various clusters.
- `collect()` combines results from each subprocess.

Parallelizing doesn’t improve run time on small data sets. Data preparation overhead is significant.

```r
system.time({
my.iris %>% group_by(Species) %>%
summarize(across(everything(), mean)) %>%
print()
})
```

```
## # A tibble: 3 x 5
##   Species       SL    SW    PL    PW
##   <fct>      <dbl> <dbl> <dbl> <dbl>
## 1 setosa      5.01  3.43  1.46 0.246
## 2 versicolor  5.94  2.77  4.26 1.33
## 3 virginica   6.59  2.97  5.55 2.03
##    user  system elapsed
##   0.024   0.000   0.023
```

```r
library(multidplyr)
cl <- new_cluster(7)
system.time({
my.iris %>% group_by(Species) %>%
partition(cl) %>%
summarize(across(everything(), mean)) %>%
collect() %>%
print()
})
```

```
## Using partial cluster of size 3
## # A tibble: 3 x 5
##   Species       SL    SW    PL    PW
##   <fct>      <dbl> <dbl> <dbl> <dbl>
## 1 setosa      5.01  3.43  1.46 0.246
## 2 versicolor  5.94  2.77  4.26 1.33
## 3 virginica   6.59  2.97  5.55 2.03
##    user  system elapsed
##   0.073   0.004   0.994
```

Timing that doesn't include data/cluster preparation shows some improvement.

```r
cl <- new_cluster(7)
system.time({
my.iris.grp <- my.iris %>%
  group_by(Species) %>% partition(cl)
my.iris.grp %>%
  summarize(across(everything(), mean)) %>%
  collect() %>%
  print()
})
```

```
## Using partial cluster of size 3
## # A tibble: 3 x 5
##   Species       SL    SW    PL    PW
##   <fct>      <dbl> <dbl> <dbl> <dbl>
## 1 setosa      5.01  3.43  1.46 0.246
## 2 versicolor  5.94  2.77  4.26 1.33
## 3 virginica   6.59  2.97  5.55 2.03
##    user  system elapsed
##   0.057   0.001   0.943
```

```r
cl <- new_cluster(7)
my.iris.grp <- my.iris %>%
  group_by(Species) %>% partition(cl)
```

```
## Using partial cluster of size 3
```

```r
system.time({
my.iris.grp %>%
  summarize(across(everything(), mean)) %>%
  collect() %>%
  print()
})
```

```
## # A tibble: 3 x 5
##   Species       SL    SW    PL    PW
##   <fct>      <dbl> <dbl> <dbl> <dbl>
## 1 setosa      5.01  3.43  1.46 0.246
## 2 versicolor  5.94  2.77  4.26 1.33
## 3 virginica   6.59  2.97  5.55 2.03
##    user  system elapsed
##   0.059   0.000   0.919
```

Parallel Computation with `sparklyr`

First, install the `sparklyr` package and a local version of Spark.

```
install.packages("sparklyr")
spark_install()
```

- Note that `spark_install` options can specify the Spark version and other options, but the defaults usually work.
- By default, Spark will then create nodes on your local machine which will use multiple cores.

The sparklyr package allows use of dplyr syntax to invoke Spark.

- spark_connect() connects to a spark session. In this case, the local session that's installed by default.
- copy_to() prepares data for use with spark.
- Always close your connection with spark_disconnect()!

```
library(sparklyr)
sc <- spark_connect(master = "local")
system.time({
my.iris.tbl <- copy_to(sc, my.iris)
my.iris.tbl %>%
  group_by(Species) %>%
  # Note: everything() doesn't work here.
  summarize(SL=mean(SL ), SW=mean(SW)) %>%
  print()
})
```

```
## # Source: spark<?> [?? x 3]
##   Species       SL    SW
##   <chr>      <dbl> <dbl>
## 1 versicolor  5.94  2.77
## 2 virginica   6.59  2.97
## 3 setosa      5.01  3.43
##     user  system elapsed
##    0.376   0.020   8.140
```

```
spark_disconnect(sc)
```

Data preparation constitutes a large chunk of this small problem’s run time.

```
sc <- spark_connect(master = "local")
system.time({
my.iris.tbl <- copy_to(sc, my.iris)
my.iris.tbl %>%
  group_by(Species) %>%
  # Note: everything() doesn't work here.
  summarize(SL=mean(SL ), SW=mean(SW)) %>%
  print()
})
```

```
## # Source: spark<?> [?? x 3]
##   Species       SL    SW
##   <chr>      <dbl> <dbl>
## 1 versicolor  5.94  2.77
## 2 virginica   6.59  2.97
## 3 setosa      5.01  3.43
##    user  system elapsed
##   0.297   0.000   7.853
```

```
spark_disconnect(sc)
```

```
sc <- spark_connect(master = "local")
my.iris.tbl <- copy_to(sc, my.iris)
system.time({
my.iris.tbl %>%
  group_by(Species) %>%
  # Note: everything() doesn't work here.
  summarize(SL=mean(SL ), SW=mean(SW)) %>%
  print()
})
```

```
## # Source: spark<?> [?? x 3]
##   Species       SL    SW
##   <chr>      <dbl> <dbl>
## 1 versicolor  5.94  2.77
## 2 virginica   6.59  2.97
## 3 setosa      5.01  3.43
##    user  system elapsed
##   0.133   0.000   1.339
```

```
spark_disconnect(sc)
```

Parallelization Summary

Summary of Parallelization Methods

Sequential	Parallelized	Type	Package
`<-`	%<-%	Assignment	`future`
`map` family	`future_map` family	Looping	`furrr`
`lapply`	`mclapply/parLapply`	Looping	`parallel`
`foreach() %do%`	`foreach` %dopar%	Looping	`doParallel`
`dplyr` cmds	`dplyr` cmds	Data Analysis	`multidplyr`
`dplyr` cmds	`dplyr` cmds	Data Analysis	`sparklyr`