future_lapply

What is future_lapply?

A function that maps a list (or vector) of data to a function and calls that function on each item in the list in parallel.

tl;dr - it's `lapply`, but can use multiple R processes

What is `lapply`?

`lapply` is a function in `base` R. If you run `?lapply` the title of the help page explains it succinctly:

> Apply a Function over a List or Vector

If you have ever used a `for` loop, this is the same concept but used as a function instead of a statement.

For example, if we wanted to build a list of the first 10 integers squared, we could do either:

```
x <- list()
for (i in 1:10) { x <- append(x, i ** 2) }
# alternatively
# for (i in 1:10) { x[[i]] = letters[[i]] }
```

or

```
y <- lapply(1:10, function(i) { i ** 2 })
```

and we'll get the same thing:

```
all.equal(x, y)
```

```
## [1] TRUE
```

Why should I care about future_lapply?

Data work often comes across problems that are embarrassingly paralell (https://en.wikipedia.org/wiki/Embarrassingly_parallel). In other words, scenarios where you need to run an identical task against a lot of different inputs and there is no dependency between the inputs.

future_lapply Examples

Prove that it works using Sleep

Simple function that does nothing by print a message to say how long it's sleeping, then wait that much time.

```r
sleep_and_print <- function(time_to_sleep = 1) {
  print(paste("Sleeping for", time_to_sleep, "seconds"))
  Sys.sleep(time_to_sleep)
}
```

This is the non-parallel version:

```r
start_time <- Sys.time()

new_data <- lapply(
  rep(1, 10), # repeat 1 ten times
  sleep_and_print
)
```

```
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
```

```r
Sys.time() - start_time
```

```
## Time difference of 10.02491 secs
```

This is the parallel version:

```r
library(future.apply)
```

```
## Warning: package 'future.apply' was built under R version 3.5.2
```

```
## Loading required package: future
```

```
## Warning: package 'future' was built under R version 3.5.2
```

```r
plan(multiprocess) ## Run in parallel on local computer using all available CPU
```

```
start_time <- Sys.time()

new_data <- future_lapply(
  rep(1, 10), # repeat 1 ten times
  sleep_and_print
)
```

```
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
## [1] "Sleeping for 1 seconds"
```

```
Sys.time() - start_time
```

```
## Time difference of 1.28902 secs
```

Cool!

Something more useful, stock prices

```
# from https://www.marketwatch.com/tools/industry/stocklist.asp?bcind_ind=9535&bcind_per
iod=3mo
stock_companies <- c(
  "VNET", "AGTK", "AKAM", "BIDU", "BCOR", "WIFI", "BRNW", "CARB", "JRJC", "CCIH", "CCOI"
, "CXDO",
  "CRWG", "EATR", "EDXC", "ENV", "FB", "FLPC", "FZRO", "GEGI", "GDDY", "IAC", "IIJI", "J
COM",
  "LOGL", "LLNW", "MOMO", "NTES", "EGOV", "OTOW", "PTOP", "SIFY", "SINA", "SMCE", "FCCN"
, "SNST",
  "TCTZF", "TCEHY", "TMMI", "TRON", "TCX", "TWTR", "XNET", "YAHOY", "YNDX", "YOOIF", "YI
PI"
)
```

Non-parallelized version:

```
start_time <- Sys.time()
stock_data <- lapply(stock_companies, quantmod::getQuote)
Sys.time() - start_time
```

```
## Time difference of 26.3874 secs
```

Now, in parallel:

```
start_time <- Sys.time()
stock_data <- future_lapply(stock_companies, quantmod::getQuote)
Sys.time() - start_time
```

```
## Time difference of 2.501605 secs
```

Getting data from the internet is a great use case for parallelization!

Something else cool, multiple SQL queries

```
library(odbc)

# this db can be recreated by running build_postgres_example_db.R
# I only tested this on Mac, but it would probably work on Linux and maybe on Windows if
 you change the host

get_query_results <- function(sql_query) {
  con <- dbConnect(
    RPostgres::Postgres(),
    host = Sys.getenv("PG_HOST"),
    user = Sys.getenv("PG_USER"),
    password = Sys.getenv("PG_PASSWORD"),
    port = Sys.getenv("PG_PORT")
  )
  results <- dbGetQuery(con, sql_query)
  dbDisconnect(con)
  return(results)
}
```

Note: This is made up data. Each table is just 1e7 (10M) letters sampled with replacement.

```
query_tables <- c("employees", "computers", "population", "currencies", "countries", "co
mpanies",
                  "stars", "cities", "rivers", "foods", "stocks", "sports")
queries <- paste("select count(*) from", query_tables)
queries
```

```
##  [1] "select count(*) from employees"  "select count(*) from computers"
##  [3] "select count(*) from population" "select count(*) from currencies"
##  [5] "select count(*) from countries"  "select count(*) from companies"
##  [7] "select count(*) from stars"      "select count(*) from cities"
##  [9] "select count(*) from rivers"     "select count(*) from foods"
## [11] "select count(*) from stocks"     "select count(*) from sports"
```

Non-parallel version:

```
start_time <- Sys.time()
query_results <- lapply(queries, get_query_results)
Sys.time() - start_time
```

```
## Time difference of 14.64563 secs
```

Parallel version:

```
start_time <- Sys.time()
query_results <- future_lapply(queries, get_query_results)
Sys.time() - start_time
```

```
## Time difference of 3.544135 secs
```

In addition to just grabbing query results, this also works well for creating multiple tables or inserting data or even selecting small batches of data from the same table. Basically, any SQL operation that doesn't depend on a logical precendence could be a good candidate.

Conculsion

Hopefully, this quick talk can help you see how running tasks that can be completed independently of each other across multiple R processes can be done easily and to much benefit using the `future.apply` package.