22 How to utilize your computer's parallel processing capabilities using `future` and `furrr`

What will this tutorial cover?

This tutorial introduces the `future` and `furrr` packages, which utilize modern computer's parallel processing capabilities to accelerate computations in R. `furrr` is constructed on top of future and employs purrr-like functions. The tutorial delves into the functions' mechanics of both packages and explores a use case of reading multiple CSV files. Additionally, we will conduct two simulations to demonstrate the functions' speed enhancement.

Who do I have to thank?

I have to thank Henrik Bengtsson and Davis Vaughan, the core developers of `future` and `furrr`, respectively. In addition, I found the tutorial video by UNMCARC on YouTube to be extremely useful in creating this tutorial. I also gained valuable insights from FXQuantTrader's response on Stackoverflow regarding how to split a vector into equally sized bins.

Speed is generally not a significant concern for the majority of R users. The R language and the Tidyverse were created with a focus on "getting things done", or as Hadley Wickham expressed on Twitter, "Tidyverse is a set of packages that try to help you do data science." (https://twitter.com/hadleywickham/status/1143546400556425218).

The user base of R reflects this philosophy as they are primarily people without formal programming education who learn the language on the job. The data sets these users typically handle are relatively small, often no larger than 2 GB in size (https://r4ds.had.co.nz/introduction.html). As data sets increase in size, code execution time becomes more significant. At this juncture, R users typically need to consider optimizing their code and utilize their machines' computational capabilities to the fullest extend possible.

Fortunately, the developers of `future` and `furrr` have designed packages that harness the parallel processing potential of modern computers. `future` simplifies the process of leveraring

these capabilities, while `furrr` builds on `future` to utilitze the parallel processing with `purrr`-like functions. One promise of `future`, as highlighted by its developer Henrik Bengtsson in a YouTube video, is that you write it once and it runs anywhere.

Before I go into the nitty-gritty, let me demonstrate an example to explain the concept more clearly. I will present two code snippets. The first one does not utilize parallel processing capabilities, while the other one utilizes all the cores of my M1 chip that is integrated into my Macbook Pro. You do not need to comprehend the code in detail at this moment, as we are primarily concerned with the output.

The subsequent code snippet calls a slow function that takes 30 seconds to complete when executed sequentially:

```
library(tictoc)
library(future)
library(furrr)

plan(sequential)

tic()
future_walk(.x = c(5, 5, 5, 5, 5, 5), .f = \(x) Sys.sleep(x))
toc()
```

```
30.043 sec elapsed
```

Now, let's compare the same computations when performed in parallel:

```
plan(multisession, workers = 6)

tic()
future_walk(.x = c(5, 5, 5, 5, 5, 5), .f = \(x) Sys.sleep(x))
toc()
```

```
5.265 sec elapsed
```

Five seconds. We managed to boost the computation speed by a factor of six.

As a general guideline for this tutorial, if you feel that some of your computations are taking too long, it may be beneficial to make use of `future` and `furrr`. However, if you are satisfied with the current speed of your computations, you need not worry about this tutorial. You can instead invest your time in learning other topics and revisit this tutorial when there is a need to speed up your computations.

22.1 An analogy on parallel processing

I am not an expert in computer hardware or parallel processing, but I'll do my best to provide you with the most comprehensive overview I can. The goal is to help you gain a better understanding of the key concepts that are essential for future and furrr.

The best explanation I found on parallel processing came from the book Introduction to High-Performance Computing (http://www.hpc-carpentry.org/hpc-chapel/11-parallel-intro/index.html) by The Carpentries. They use the example of painting a wall to illustrate the concept. Let's say you have four walls to paint, which is the problem at hand. Now, you can break this down into four individual tasks, painting each wall. In theory, these tasks are independent, meaning you can either paint the walls one after another, or in parallel. We refer to these four tasks as "concurrent," which just describes the property of a task, and doesn't necessarily mean that they are executed in parallel or sequentially. It all depends on the resources available. For instance, if you are painting the walls alone, the tasks are concurrent, but you need to work sequentially. However, with two painters, you can execute the concurrent tasks in parallel.

So, the painters are what we call **workers** in parallel processing. Generally, you can assign the number of workers equivalent to the number of cores present in your machine. For instance, my personal laptop is a MacBook Pro with an M1 chip that has eight cores. This means that I can create at least eight workers to operate in parallel. The term worker will become important in just a moment when we begin to use the `plan()` function of `future`.

22.2 The future package

The cross-platform `future` package is a high-level API that allows non-technical people to take advantage of parallel processing capabilities when working in R. The package was developed by Henrik Bengtsson, an assistant professor of epidemiology and biostatistics at the University of California. The first version of `future` was released on GitHub in 2015. At the time of writing this tutorial, `future` has reached version 1.31.0.

`future` works in three steps:

- Step 1: You need to decide how you want to parallelize your code. You have four options: running the code sequentially, running in multiple R sessions, running on multiple cores, or running on a cluster of multiple machines. You can use the `plan()` function to make this decision. Note that when you work in R-Studio, using multiple cores is not recommended and doesn't work on Windows, so most of the time, you will work in multiple R-Sessions.
- Step 2: Once you've decided how to parallelize your code, you need to determine which part of your code should be run in parallel. You can use the `future()` function to make this decision.

- Step 3: You need to retrieve the value of your computations run in future. You can use the `value()` function to get these values.

Let's use my laptop as an example again. I'm currently using an Apple MacBook Pro with an M1 chip that has 8 cores. If I didn't know the number of cores, I could use the `availableCores()` function to check it:

```
library(future)
availableCores()
```

```
system
     8
```

To decide whether you want to run your code sequentially or in parallel, you use the `plan()` function. By default, when you load the `future` package, the code runs sequentially. Here's an example of setting up a sequential process. The following code computes the square of three numbers using a slow function called `slow_function_square_number`:

```
plan(sequential)

# This function takes more than 2 seconds to run
slow_function_square_number <- function(i) {
  Sys.sleep(2)
  i^2
}

# start timer
tic()

num_one <- slow_function_square_number(2)
num_two <- slow_function_square_number(4)
num_three <- slow_function_square_number(9)

num_one
```

```
[1] 4
```

```
num_two
```

```
[1] 16
```

```
num_three
```

```
[1] 81
```

```
# end timer
toc()
```

```
6.033 sec elapsed
```

Take a look at `plan(sequential)` and the function calls to `slow_function_square_number()`. This tells the computer to not use the parallel processing capabilities and instead run the computations one after another. Keep in mind that each call of `slow_function_square_number()` takes at least 2 seconds to run because the function was intentionally delayed by two seconds.

To run the computations in parallel, we need to make the following changes:

- First, let's change the `plan` and use `plan(multisession)`. This plan will create multiple R-Sessions that run in parallel. By default, this function uses as many workers as are available.
- Second, we'll wrap the three function calls `slow_function_square_number()` inside `future()`. By doing so, we're telling future that these computations should run in parallel based on the number of sessions we specified in `plan(multisession)`.
- Third, we'll retrieve the values of our `future()` computations using `value()`.
- Fourth, we'll call `plan(sequential)` to close the multisession workers. This is considered good practice in the future package.

```
plan(multisession)

# This function takes more than 2 seconds to run
slow_function_square_number <- function(i) {
  Sys.sleep(2)
  i^2
}

# start timer
tic()

num_one <- future({slow_function_square_number(2)})
num_two <- future({slow_function_square_number(4)})
num_three <- future({slow_function_square_number(9)})
```

```
  value(num_one)
```

```
[1] 4
```

```
  value(num_two)
```

```
[1] 16
```

```
  value(num_three)
```

```
[1] 81
```

```
  # end timer
  toc()
```

```
2.124 sec elapsed
```

```
  plan(sequential)
```

We have achieved a threefold increase in speed. Now, this is the explicit way of using `future` where the functions `future()` and `value()` are explicitly named. Alternatively, once can use `future()` implicitly through the use of the `%<-%` operator. Instead of the explicitly declaring `f <- future({ exp})`, one writes `f %<-% {exp}`:

```
  plan(multisession)

  # This function takes more than 2 seconds to run
  slow_function_square_number <- function(i) {
    Sys.sleep(2)
    i^2
  }

  # start timer
  tic()

  num_one %<-% {slow_function_square_number(2)}
```

```
num_two %<-% {slow_function_square_number(4)}
num_three %<-% {slow_function_square_number(9)}

# end timer
toc()
```

```
0.097 sec elapsed
```

```
plan(sequential)
```

When is this technique useful? One scenario is when there is a need to read multiple files into memory. For example, if you have numerous CSV files stored on your disk and would like to read them in parallel. In the code below, I generate 5000 CSV files and store them in a folder named `diamonds_data`:

```
library(fs)

dir_create("diamonds_data")
diamonds_samples <- map(1:1000, ~ slice_sample(diamonds, n = 4000))
iwalk(diamonds_samples, ~ write_csv(., paste0("diamonds_data/", .y, ".csv")))
```

Then, using the `dir_ls` function from the `fs` package, I retrieve the path names of these files:

```
csv_files <- dir_ls(path = "diamonds_data/", glob = "*.csv", type = "file")
csv_files |> head()
```

```
diamonds_data/1.csv    diamonds_data/10.csv   diamonds_data/100.csv
diamonds_data/1000.csv diamonds_data/101.csv  diamonds_data/102.csv
```

First, I execute the non-parallelized slow version:

```
plan(sequential)

tic()
diamonds_data_complete_slow <- purrr::map(.x = csv_files,
                                          .f = \(file_path) read_csv(file_path) |>
                                            mutate(name = file_path)) |>
  list_rbind()
toc()
```

```
29.519 sec elapsed
```

To expedite this computation, a few modifications are necessary. Firstly, we need to divide the dataset into bins. This is done using the function call `split(csv_files, rep_len(1:3, length(csv_files)))`, which is a helpful tip I discovered on Stackoverflow. I would like to give credit to FXQuantTrader for sharing it. Following this, I execute `plan(multisession)` and use three implicit `future` calls, one for each bin, and then execute `map()`. Finally, the three lists are combined into a dataframe using `list_rbind`:

```
# Split CSV paths into a list of length
# three. Each list element holds roughly the same number of CSV paths
# Tip: https://stackoverflow.com/questions/3318333/split-a-vector-into-chunks
csv_files_splits <- split(csv_files, rep_len(1:3, length(csv_files)))

plan(multisession)

tic()

diamonds_data_one %<-% purrr::map(.x = csv_files_splits[[1]],
                                  .f = \(file_path) read_csv(file_path)
                                  |> mutate(name = file_path))

diamonds_data_two %<-% purrr::map(.x = csv_files_splits[[2]],
                                  .f = \(file_path) read_csv(file_path)
                                  |> mutate(name = file_path))

diamonds_data_three %<-% purrr::map(.x = csv_files_splits[[3]],
                                    .f = \(file_path) read_csv(file_path)
                                    |> mutate(name = file_path))


# Combine data frames into one data frame
diamonds_data_complete_fast <- bind_rows(
  diamonds_data_one |> list_rbind(),
  diamonds_data_two |> list_rbind(),
  diamonds_data_three |> list_rbind()
  )

toc()
```

```
14.371 sec elapsed
```

```
plan(sequential)

# Remove data sets
rm(diamonds_data_one)
rm(diamonds_data_two)
rm(diamonds_data_three)
```

Executing this code takes only 15 seconds. It is worth noting that I have repeated my code thrice, which is not ideal coding practice. Rather than this approach, I could have implemented a loop. Fortunately, there is an alternative solution available in the form of the `furrr` package.

22.3 The `furrr` package

`furrr`, created and maintained by Davis Vaughan, is built on top of future. Davis is a Software Engineer at R-Studio and started contributing to the package in 2018, releasing the first version on GitHub. The primary concept behind `furrr` is to harness the capabilities of the `future` package with purrr-like functions. The idea is straightforward: `furrr` includes the typical purrr functions for iterations, each of which is prefixed with `future_`:

- `map` -> `future_map`
- `map2` -> `future_map2`
- `pmap` -> `future_pmap`
- `walk` -> `future_walk`
- `walk2` -> `future_walk2`
- `pwalk` -> `future_pwalk`

The furrr functions are intended to be a "near drop in replacement for purrr functions" (https://furrr.futureverse.org/). The developers have made it remarkably straightforward to combine `furrr` with `future`. You need to ensure two things. First, you must still define the `plan()` and specify whether you want to execute the computation sequentially, in multisession, or in multicore. Second, you must replace your `purrr` function with the corresponding `furrr` function. That's all there is to it.

We previously increased the speed of reading our files by a factor of five by splitting the CSV file paths into bins and using implicit `future()` calls. With `furrr`, the code now looks as follows:

```
library(furrr)

plan(multisession)
```

```
tic()
diamonds_data_complete_fast_futuremap <- future_map(
  .x = csv_files,
  .f = \(file_path) read_csv(file_path) |>
    mutate(name = file_path)
) |>
  list_rbind()
toc()
```

```
14.813 sec elapsed
```

```
plan(sequential)
```

Again, the code takes 15 seconds to execute. The advantage is that the code is significantly more straightforward to comprehend, and we no longer need to contemplate how to utilize the parallel processing abilities of `future`.

22.4 Simulating the speed of `furrr`

To fully appreciate the time-saving benefits of `furrr` and `future`, we can conduct a simulation using various parameters. In this simulation, I once again use the same code as before to load numerous CSV files into memory, but with varying two parameters: the number of cores for the multisession and the number of files that I will read into `future_map()`. Rather than delving into the intricacies of the code, I want to discuss the simulation's results instead.

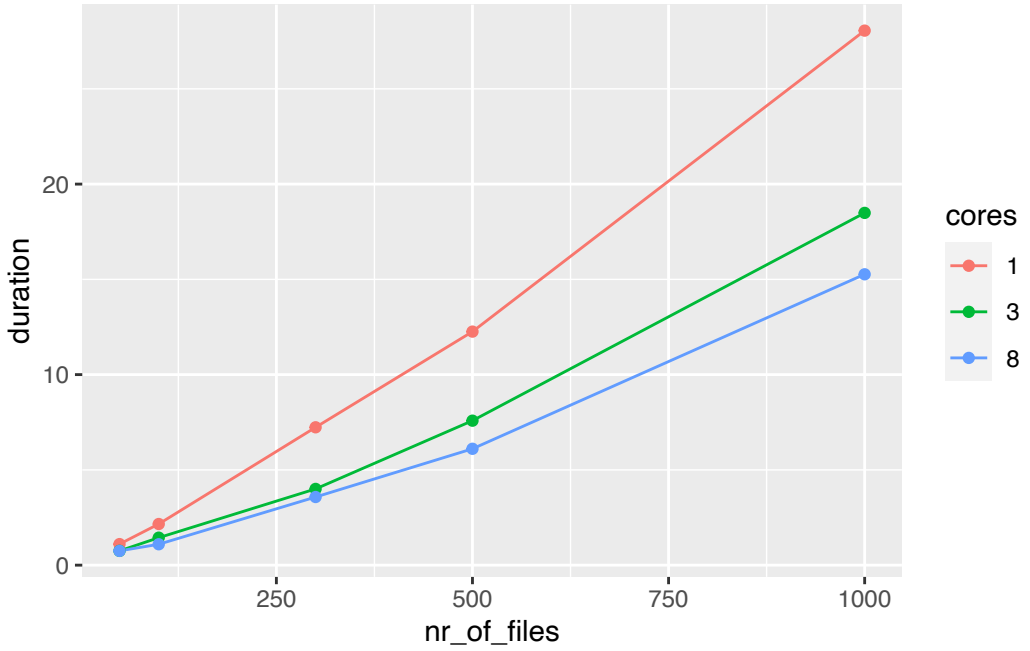
```
simulation_data <- crossing(cores = c(1, 3, 8),
                            nr_of_files = c(50, 100, 300, 500, 1000))

res <- pmap(
  .l = simulation_data,
  .f = \(cores, nr_of_files) {

    plan(multisession, workers = cores)

    tic()
    temp_res <- future_map(
      .x = csv_files |> sample(nr_of_files, replace = TRUE),
      .f = \(file_path) read_csv(file_path) |>
        mutate(name = file_path)
```

```
    ) |>
      list_rbind()

    time_results <- toc()

    plan(sequential)
    rm(temp_res)

    tibble(
      cores = cores,
      nr_of_files = nr_of_files,
      duration = time_results$toc - time_results$tic
    )
  }
) |> list_rbind()
```

```
1.324 sec elapsed
2.553 sec elapsed
7.555 sec elapsed
13.245 sec elapsed
28.694 sec elapsed
0.759 sec elapsed
1.29 sec elapsed
4.101 sec elapsed
7.588 sec elapsed
18.688 sec elapsed
0.764 sec elapsed
1.071 sec elapsed
3.114 sec elapsed
5.683 sec elapsed
15.084 sec elapsed
```

The simulation results are stored in the `res` data frame. We can visualize the results with ggplot2:

```
res |>
  mutate(cores = as_factor(cores)) |>
  ggplot(aes(x = nr_of_files, y = duration,
             color = cores)) +
  geom_line() +
  geom_point()
```



It turns out that the more files that have to be read into the memory, the longer the calculations take. This is no surprise. We also see that the calculations are always faster with more cores. The other finding is that using three or eight cores is massively more efficient than using one core. However, it doesn't make too much difference whether you use 3 or 8 cores.

One might assume that `furrr` provides the same computational benefits when used with nested data. However, when working with grouped and nested data, `furrr` is actually slower than the `purrr` functions. To illustrate this, consider the following simulation. We generate datasets of different sizes, nest the simulated data, and compute linear models for each nested dataset:

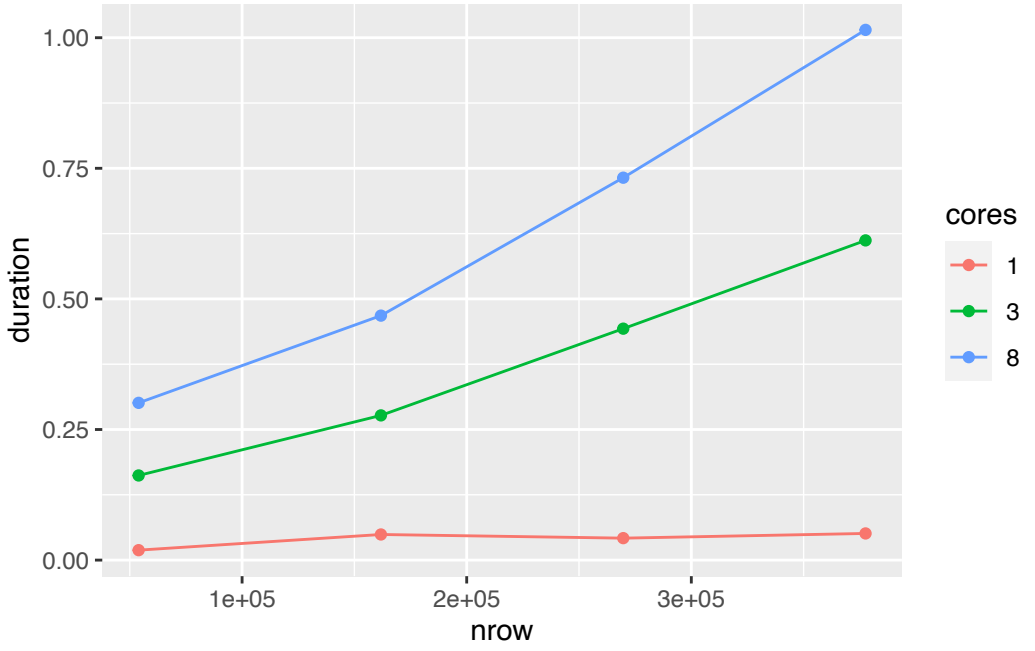
```
simulation_data <- crossing(cores = c(1, 3, 8), bootstraps = c(1, 3, 5, 7))

res_nested <- pmap(
  .l = simulation_data,
  .f = \(cores, bootstraps) {

    plan(multisession, workers = cores)


    bootstrapped_data <- map(1:bootstraps,
                             \(x) slice_sample(diamonds,
                                               prop = 1, replace = TRUE)) |>
```

```
      list_rbind()

    tic()


    simulation_res <- bootstrapped_data |>
      nest(data = -cut) |>
      mutate(
        model = future_map(.x = data,
                           .f = \(x) lm(x ~ z, data = x))
      )

    time_results <- toc()

    plan(sequential)

    tibble(
      cores = cores,
      duration = time_results$toc - time_results$tic,
      nrow = nrow(bootstrapped_data)
    )

  }
) |> list_rbind()
```

```
0.019 sec elapsed
0.049 sec elapsed
0.042 sec elapsed
0.051 sec elapsed
0.162 sec elapsed
0.277 sec elapsed
0.443 sec elapsed
0.612 sec elapsed
0.301 sec elapsed
0.468 sec elapsed
0.732 sec elapsed
1.015 sec elapsed
```

```
res_nested |>
  mutate(
    cores = as.factor(cores)) |>
```

```
mutate(cores = as_factor(cores)) |>
ggplot(aes(x = nrow, y = duration,
           color = cores)) +
geom_line() +
geom_point()
```



Ironically, utilizing only one worker is the fastest method. This is a well-known caveat in `furrr` and may be related to the issue of data transfer. More information about this issue can be found here: https://furrr.futureverse.org/articles/gotchas.html#grouped-data-frames. I would like to provide you with a solution to this problem, but regrettably, I have not discovered one yet. Even chatGPT didn't find a workaround (Prompt: "Can you rewrite this code so it actually runs faster with furrr").

Summary

- `future` is a high-level API that can tap into the parallel processing capabilities of modern computers. `furrr` is build on top of `future` by using purrr-like functions to speed up iterations in R.
- If you want to use `future`, there are three simple steps you need to follow: First, you need to set a `plan()`. Then, you wrap the computations that you want to run in parallel in `future()`. Finally, you retrieve the results using `value()`.

- You can run `future` explicitly by using `future()` and)`value()`, or implicitly by using the `%<-%` operator.
- If you want to measure how fast your computations are running, you can use the `tic` and `toc` functions from the `tictoc` package.
- While `furrr` is a great tool, it doesn't work well with grouped and nested data. In fact, it can even run slower than `purrr` functions in certain cases.