Futureverse - A Unifying Parallelization Framework in R for Everyone

STATS/BIODS 352: Topics in Computing for Data Science, Bridging Methodology and Practice, Stanford University, 2023

Henrik Bengtsson

2023-05-08

A future is a programming construct designed for concurrent and asynchronous evaluation of code, making it particularly useful for parallel processing. The future package implements the Future API for programming with futures in R. This minimal API provides sufficient constructs for implementing parallel versions of well-established, high-level map-reduce APIs. The future ecosystem supports exception handling, output and condition relaying, parallel random number generation, and automatic identification of globals lowering the threshold to parallelize code. The Future API bridges parallel frontends with parallel backends, following the philosophy that end-users are the ones who choose the parallel backend while the developer focuses on what to parallelize. A variety of backends exist, and third-party contributions meeting the specifications, which ensure that the same code works on all backends, are automatically supported. The lectures focus on R but programmers from other languages will also find the material useful.

Table of contents

Introduction 5

Take-home messages of this lecture set 6

Disclaimer 7

I Lecture 1 8

1 Why do we parallelize? 9

2 How do you do two things at the same time in R? 10

3 Parallelizing map-reduce calls 13
3.1 Parallelizing a map-reduce call using the 'future.apply' package 14
3.2 Parallelizing a map-reduce call using the 'furrr' package 15
3.3 Comment about pipes 16

4 For loops - can they be parallelized? 17
4.1 Parallelize a for-loop using futures 17
4.2 Replace a for-loop with a map-reduce call 18
4.3 How do I know if my for-loop can be rewritten as a map-reduce call? 20
4.3.1 Not all algorithms can be parallelized 21

II Lecture 2 22

5 Demo and parallel backends 23
5.1 Parallelize on local machine 23
5.2 Parallelize on local machine 23
5.2.1 Standalone background R processes 23
5.2.2 Forked R processes 24
5.3 Parallelize on multiple machines 24
5.4 Parallelize via HPC job scheduler 25
5.5 Alternatives 26

6 The Future API **27**
6.1 Three atomic building blocks . . . 28
6.1.1 Mental model: The Future API decouples a regular R assignment into two parts . . . 28
6.1.2 Keep doing other things while waiting . . . 31
6.1.3 Evaluate several things in parallel . . . 31
6.2 Choosing a parallel backend . . . 34
6.2.1 sequential (default) . . . 34
6.2.2 multisession: in parallel on local computer . . . 35
6.2.3 cluster: in parallel on multiple computers . . . 35
6.2.4 There are other parallel backends and more to come . . . 36
6.3 Revisiting the future assignment operator (`%<-%`) . . . 37

7 Non-exportable objects **38**
7.1 A database connection is only valid in the current R session . . . 39
7.2 Same problem when saving to file . . . 40
7.3 Workaround . . . 41
7.4 Futureverse can help us detect this before it happens . . . 42
7.5 What about forked parallelization? . . . 45

8 foreach() is not a for-loop **47**
8.1 Super assignment (`<<-`) is not a solution . . . 47
8.2 Return instead of assign in map-reduce calls . . . 49
8.3 foreach() is a map-reduce function . . . 51

9 mclapply() - is it really magic? **54**
9.1 Output is at best fake from mclapply() . . . 54
9.2 Warnings are lost by mclapply() . . . 55
9.3 Errors are mangled . . . 56
9.4 What happens when a parallel crashes? . . . 58

Summary **60**

III Appendix **61**

References **62**

Introduction

In these lectures, I will cover how to parallelize R code using the Futureverse, which at its core consist of the **future** package (Bengtsson (2021)). Other packages in the Futureverse build on the **future** package to provide more powerful features, e.g. **future.apply**, **furrr**, and **doFuture**.

The Futureverse builds upon, enhances, and unifies established parallelization frameworks in R, e.g. **parallel** and **foreach**. You can think of it as a user friendly, unifying wrapper on top of many of the existing more low-level alternatives that each come with their own unique functions and settings. By using Futureverse, there are less things you have to worry about and your code will be less cluttered by special parallelization instructions.

The **future** package was introduced in 2015, and is now a stable and well established solution for parallelization in R. For example, it is among the top-0.9% most downloaded R packages, and there are hundreds of R packages that use it for their parallelization needs.

Take-home messages of this lecture set

By following these two lectures, you will learn that:

- parallelization does not have to be hard
- there are things you *cannot* parallelize
- Futureverse simplifies parallelization in R (disclaimer!)
- `foreach()` is *not* the same as a for-loop
- *forked* parallel processing is neat, but we should use it with great caution

You will also learn:

- a bit about the "future" concept for parallel programming
- why it is called "futures"
- that many programming languages supports futures, e.g. R, Python, Julia, and C++
- about common mistakes to avoid

From this, I hope that you will think of parallelization as being less magic, especially if you never used it before.

Disclaimer

I am the creator and lead maintainer of the Futureverse ecosystem. I choose to use it to teach parallelization in R, because I think it is the simplest way to parallelize tasks in R.

Part I

Lecture 1

1 Why do we parallelize?

Parallel & distributed processing can be used to:

- speed up processing (wall time)
- lower memory footprint (per machine)
- avoid data transfers (compute where data lives)
- other reasons, e.g. asynchronous user interface

We may choose to parallelize on:

- Your personal laptop or work desktop computer (single user)
- A shared powerful user (multiple users)
- High-performance compute (HPC) cluster (multiple users) with a job scheduler, e.g. Slurm, Son of Grid Engine (SGE)

2 How do you do two things at the same time in R?

```
Attaching package: 'listenv'
```

```
The following object is masked from 'package:purrr':

    map
```

Imagine we have a very slow function called `slow_sum()` that takes a numeric vector as input, calculates the sum, and returns it as numeric scalar:

```
slow_sum <- function(x) {
  sum <- 0

  for (value in x) {
    Sys.sleep(1.0)  ## one-second slowdown per value
    sum <- sum + value
  }

  sum
}
```

For example, we can calculate the sum of $1, 2, ..., 10$ as:

```
y <- slow_sum(1:10)
y
```

```
[1] 55
```

The problem is that this takes more than 10 seconds to complete, e.g.

```
tic()
y <- slow_sum(1:10)
toc()
```

```
Time difference of 10.1 secs
```

This will be costly if we want repeat this twice or more;

```
tic()
y1 <- slow_sum(1:10)
y2 <- slow_sum(11:20)
toc()
```

```
Time difference of 20.2 secs
```

Wouldn't it be great if we could run these two tasks concurrently?

If they could run at the same time, we would finish both in the same period of time as when we call the function once. It turns out we can use the **future** package for this. Here's is how we can do it with a minimal tweak.

```
library(future)     ## defines %<-%
plan(multisession)  ## set them to run in parallel

y1 %<-% slow_sum(1:10)
y2 %<-% slow_sum(11:20)
y1
```

```
[1] 55
```

```
y2
```

```
[1] 155
```

The `%<-%` assignment operator works by launching `slow_sum(1:10)` in the background, preparing to assign the result to `y1` when its done, and then returning immediately. Same for the second expression. This means that both of these *future assignments* complete almost instantly:

```
tic()
y1 %<-% slow_sum(1:10)
y2 %<-% slow_sum(11:20)
toc()
```

```
Time difference of 0.9 secs
```

What happens next, is that whenever we try to "use" the value of `y1` or `y2`, R will automatically wait for the result to become available. This is where we might have to wait:

```
y1
```

```
[1] 55
```

```
toc()
```

```
Time difference of 9.6 secs
```

In other words, we have to wait for `y1` to complete, but, since the both *future expressions* ran in parallel, `y2` completes in about the same time, and we do *not* have to spend time waiting for its result:

```
y2
```

```
[1] 155
```

```
toc()
```

```
Time difference of 11.4 secs
```

So, all in all, we completed both tasks in the same amount of time as as single one.

3 Parallelizing map-reduce calls

Next, assume we have four sets of numeric vectors, and we want to calculate `slow_sum()` for each of them. We have them in a list, e.g.

```
xs <- list(1:25, 26:50, 51:75, 76:100)
```

We could keep doing what we did in the previous section;

```
ys <- list()
ys[[1]] <- slow_sum(xs[[1]])
ys[[2]] <- slow_sum(xs[[2]])
ys[[3]] <- slow_sum(xs[[3]])
ys[[4]] <- slow_sum(xs[[4]])
```

This will give us the results in a list `ys` of the same length as `xs`, e.g.

```
str(ys)
```

```
List of 4
 $ : num 325
 $ : num 950
 $ : num 1575
 $ : num 2200
```

This approach will become very tedious when there are more sets, i.e. when `length(xs)` is large. It is also *error prone*, e.g. it's too easy to introduce a silent bug from a single typo, e.g.

```
ys <- list()
ys[[1]] <- slow_sum(xs[[1]])
ys[[2]] <- slow_sum(xs[[2]])
ys[[3]] <- slow_sum(xs[[2]])
ys[[4]] <- slow_sum(xs[[4]])
```

Whenever you find yourself repeating code by cut'n'paste from previous lines, it's a good indicator to stop and think. There's almost always a better way to this - you just have find what it is!

R is designed to simplify above type of tasks. In this case we can use `lapply()` to achieve the same:

```
ys <- lapply(xs, slow_sum)
str(ys)
```

```
List of 4
 $ : num 325
 $ : num 950
 $ : num 1575
 $ : num 2200
```

3.1 Parallelizing a map-reduce call using the 'future.apply' package

Since there are four sets of data, each comprise of 25 values, and each value takes about one second to process, processing all of the data takes about 100 seconds;

```
tic()
ys <- lapply(xs, slow_sum)
toc()
```

```
Time difference of 100.6 secs
```

Can we speed this up by processing the different elements in `xs` concurrently?

Yes, we can. Unfortunately, the built-in `lapply()` function is not implemented to run in parallel. However, the **future.apply** package provides the `future_lapply()` function that can run in parallel. It is designed to be a plug-and-play replacement of `lapply()`. We just have to prepend `future_` to the `lapply` name.

```
library(future.apply)
plan(multisession)

ys <- future_lapply(xs, slow_sum)
str(ys)
```

```
List of 4
 $ : num 325
 $ : num 950
 $ : num 1575
 $ : num 2200
```

By design, this gives identical result to `lapply()`, but it performs `slow_sum(xs[[1]])`, `slow_sum(xs[[2]])`, ..., in parallel.

To convince ourselves it runs in parallel, be can measure the processing time:

```
tic()
ys <- future_lapply(xs, slow_sum)
toc()
```

```
Time difference of 26.2 secs
```

3.2 Parallelizing a map-reduce call using the 'furrr' package

If you use the Tidyverse framework, you might already be aware of the **purrr** package. It provides an alternative to the built-in `lapply()` function called `map()`. It works very similarly. Our

```
ys <- lapply(xs, slow_sum)
```

can be written as:

```
library(purrr)

ys <- map(xs, slow_sum)
```

It gives identical results. To run this in parallel, you can use `future_map()` of the **furrr** package. Just as `future_lapply()` can replace `lapply()` as-is, `future_map()` replaces `map()` as-is:

```
library(furrr)
plan(multisession)

ys <- future_map(xs, slow_sum)
```

3.3 Comment about pipes

All of the above works also with *pipes*. You can use the semi-legacy **magrittr** `%>%` pipe operator popularized by Tidyverse, or the zero-cost `|>` pipe operator that is now built-in with R.

Just like the Tidyverse maintainers, I recommend using the latter. There is zero overhead added when using it, and there is truly no extra code being executed behind the scenes. Instead, it just a different way that R parses you code - after that everything is the same. The following two R expressions are *identical* from R's perspective:

```
y <- g(f(x))
```

and

```
y <- x |> f() |> g()
```

The analogue is mathematics, is that the following expressions are equivalent:

$$h(x) = g(f(x))$$

$$h(x) = (f \circ g)(x)$$

Thus, when programming in R, we can use either of:

```
ys <- lapply(xs, slow_sum)
ys <- xs |> lapply(slow_sum)
```

Same for

```
ys <- future_lapply(xs, slow_sum)
ys <- xs |> future_lapply(slow_sum)
```

and

```
ys <- future_map(xs, slow_sum)
ys <- xs |> future_map(slow_sum)
```

4 For loops - can they be parallelized?

Let's go back to your example with:

```
xs <- list(1:25, 26:50, 51:75, 76:100)
```

where we used:

```
ys <- lapply(xs, slow_sum)
```

to calculate the the `slow_sum()` of each element in `x`.

Someone, who is not familiar with R and its `lapply()`, might choose to solve this problem using a for-loop, e.g.

```
ys <- list()
for (ii in 1:length(xs)) {
  ys[[ii]] <- slow_sum(xs[[ii]])
}
```

By the way, can you spot the potential problem with this solution?

When using `1:length(xs)`, there is a risk it becomes `1:0` (= `c(1, 0)`), which happens if `xs` is an empty list. To avoid this, it's safer to use `seq_len(length(xs))`, or the short cut `seq_along(xs)`. Those will return an empty index vector, if `length(xs) == 0`.

4.1 Parallelize a for-loop using futures

Since we are already familiar with the future assignment operator (`%<-%`), we can parallelize the above for-loop as follows:

```
library(future)
library(listenv)
plan(multisession)
```

```
ys <- listenv()
for (ii in seq_along(xs)) {
  ys[[ii]] %<-% slow_sum(xs[[ii]])
}
ys <- as.list(ys)
```

Without going into the technical details, we cannot use a list for `ys` when doing this. Instead, we need to use a *list environment* part of the **listenv** package. Then, at the very end, we *coerce* it to a regular list using `as.list()`. This gives identical results, but the for loop is run in parallel.

This works, but if `length(xs)` is large, it is not as efficient as parallel map-reduce solutions such as `future_lapply()` and `future_lap()`.

4.2 Replace a for-loop with a map-reduce call

Regardless of sequential or parallel processing, I recommend to always strive toward using map-reduce functions instead of for-loop. It is not always possible to do so, but in many cases it is.

If we look at the above for-loop, we see that each iteration is independent of the others, e.g. the result from the second iteration is independent of the result obtained in the first iteration, i.e. the value of `ys[[2]]` only depends on `xs[[1]]` and the function `slow_sum()`, but not on `ys[[1]]` or any other `xs` elements. We will return to this later, but when a for-loop algorithm has this property, because turn it into a map-reduce call, and we already know that we can parallelize those.

As a rule of thumb, the first step towards replacing a for-loop with a map-reduce call, is to get rid of the auxiliary index variable, which in our example is `ii`. In our example, all we use `ii` for is to extract element `xs[[ii]]` and then assign the result to `ys[[ii]]`. We can make this explicit by rewriting the for-loop as:

```
ys <- list()
for (ii in seq_along(xs)) {
  x <- xs[[ii]]
  y <- slow_sum(x)
  ys[[ii]] <- y
}
```

This form helps us see how `ii` is used and that there is no dependencies between iterations.

Next, we can replace the above *iterate-over-indices* approach with an *iterate-over-elements* approach, as in:

```
ys <- list()
for (x in xs) {
  ys <- c(ys, slow_sum(x))
}
```

Note how we got rid of the iteration index `ii`. This might look like *syntactic sugar*, but it's an important move as we will see soon. First, by getting rid of the index variable, the code becomes "cleaner". By "cleaner" we often mean it is more readable (to the used reader). Second, cleaner code is often also less error prone. For example, without the index variable, there is no risk we're making mistakes using it, e.g. we might use `ii + 1` when it should be `ii`.

The above code reads as "for each element `x` in the list `xs`, do …". Note how similar this is to "for each element `x` in the list `xs`, apply function …', which is how a map-reduce call reads.

Since we got rid of `ii`, we can replace the latter for-loop with an `lapply()` call as in:

```
ys <- lapply(xs, slow_sum)
```

Digest that for a while, and note how concise and clear that is compare that the more explicit for-loop that we started out with:

```
ys <- list()
for (ii in 1:length(xs)) {
  ys[[ii]] <- slow_sum(xs[[ii]])
}
```

One way to think about it is that:

- an `lapply()` call communicates *what* is done, whereas
- a for-loop communicates *how* it is done.

After working with R for a while, and seeing a lot of `lapply()` calls, you will get used to this form and prefer to to the more verbose syntax that comes from using a for-loop. I've worked with R for more than 20 years and to me an `lapply()` call is much easier to read and brings much less mental load compared to a for-loop. It is only if I try to think about what happens under the hood of `lapply()`, that it can become overwhelming. I can imagine there is some kind of for loop running inside, but if you think about it, it does not matter how it is implemented internally.

Finally, we now have a `lapply()` call, and we already know how to parallelize that; all we have to do is replace `lapply()` with `future_lapply()`.

Conclusion: To parallelize a for-loop, convert it first to a map-reduce call, and then replace that with the corresponding `future_` version.

4.3 How do I know if my for-loop can be rewritten as a map-reduce call?

Not all for-loops can be parallelized. We already touched upon one rule of thumb above. If the different iterations are independent of each other, we can get rid of iteration index variable (`ii`) and then we can basically always parallelize the code.

Another "smell test" for being able to parallelize a for-loop comes from asking ourselves the following question:

Q. Does it matter in what order we "process" the different elements in `xs`, i.e. in what order we call `slow_sum()` on each of them?

If the answer is "No, it doesn't matter", then we can parallelize the for loop.

For example, if we go back to our original for-loop;

```
ys <- list()
for (ii in 1:length(xs)) {
  ys[[ii]] <- slow_sum(xs[[ii]])
}
```

we note that we could *iterate* over the elements in `xs` in reverse order:

```
ys <- list()
for (ii in length(xs):1) {
  ys[[ii]] <- slow_sum(xs[[ii]])
}
```

and still get the identical result. We could even iterate over the elements in a random order, e.g.

```
ys <- list()
for (ii in sample(1:length(xs), replace = FALSE)) {
  ys[[ii]] <- slow_sum(xs[[ii]])
}
```

In all three cases, `ys` holds identical values in the same order.

By the way, a similar smell test for an `lapply()` call is to check if we get the same results if we reverse the input, and the output, as in:

```
ys_r <- lapply(rev(xs), slow_sum)
ys <- rev(ys_r)
```

If you think about it, `lapply()` could very well be implemented such that it process the elements in reverse order. If you read the documentation, `help("lapply")`, there is nothing saying in what order the elements is processed. This is intentional, because we should not use this function under the assumption that elements are process in a certain order. This is what `future_lapply()` and friends rely on and why it is valid to parallelize map-reduce calls.

Conclusion: If we have an algorithm that allows us to reverse the order of the processing, or process the elements in a random order, we can most likely also run the iterations concurrently.

4.3.1 Not all algorithms can be parallelized

It is only for some algorithms that the order of the processing does not matter. We referred to such algorithms as *embarrassingly parallelizable algorithms*. As the term suggests, the algorithm can be parallelized, i.e. it is valid to process the elements concurrently. For other algorithms, it is essential that the elements are processed in a given order. It is often that the algorithm is such that the input data in one iteration depends on the output of the previous iteration. We cannot use parallelization for such algorithms.

Q: What about the following algorithm - can it be reversed?

```
ys <- list()
y <- 0
for (ii in 2:length(xs)) {
  x <- xs[[ii]]
  y <- ys[[ii - 1]]
  ys[[ii]] <- slow_sum(x + y)
}
```

Without going into details, another criteria is that `slow_sum()` does not have, so called, *side effects*. This is now mathematical functions work, and this is one reason why we refer to R as a functional language. As a rule of thumb, most function calls in R does not have side effects, and if they do, you often already know it.

Part II

Lecture 2

5 Demo and parallel backends

5.1 Parallelize on local machine

```
library(future)
options(future.demo.mandelbrot.resolution = 5000)   # 5000x5000 px

demo("mandelbrot", ask = FALSE)
```

1. `plan(sequential)` - default
2. `plan(multisession)`
3. `plan(multicore)`

5.2 Parallelize on local machine

5.2.1 Standalone background R processes

```
plan(multisession)
plan(multisession, workers = availableCores())  ## default
plan(multisession, workers = 2)
```

Real-world example:

```
library(future.apply)
plan(multisession, workers = 3)

info <- future_lapply(seq_len(nbrOfWorkers()), function(idx) {
  data.frame(idx = idx, hostname = Sys.info()[["nodename"]], pid = Sys.getpid())
})
info <- do.call(rbind, info)
info
```

```
  idx   hostname    pid
1   1 hb-x1-2023 252154
2   2 hb-x1-2023 252153
3   3 hb-x1-2023 252152
```

5.2.2 Forked R processes

```
plan(multicore)
plan(multicore, workers = availableCores())  ## default
plan(multicore, workers = 2)
```

```
library(future.apply)
plan(multicore, workers = 3)

info <- future_lapply(seq_len(nbrOfWorkers()), function(idx) {
  data.frame(idx = idx, hostname = Sys.info()[["nodename"]], pid = Sys.getpid())
})
info <- do.call(rbind, info)
info
```

```
  idx   hostname    pid
1   1 hb-x1-2023 252285
2   2 hb-x1-2023 252286
3   3 hb-x1-2023 252287
```

5.3 Parallelize on multiple machines

```
plan(cluster)
plan(cluster, workers = availableWorkers())  ## default
plan(cluster, workers = c("dev1", "dev2", "dev3"))
```

Real-world example:

```
library(future.apply)
plan(cluster, workers = c("dev1", "dev2", "dev3", "dev3", "dev3"))

info <- future_lapply(seq_len(nbrOfWorkers()), function(idx) {
```

```
  data.frame(idx = idx, hostname = Sys.info()[["nodename"]], pid = Sys.getpid())
})
info <- do.call(rbind, info)
info
```

```
  idx             hostname    pid
1   1 dev1.wynton.ucsf.edu 281122
2   2 dev2.wynton.ucsf.edu  29646
3   3 dev3.wynton.ucsf.edu  43826
4   4 dev3.wynton.ucsf.edu  43881
5   5 dev3.wynton.ucsf.edu  43921
```

5.4 Parallelize via HPC job scheduler

```
plan(batchtools_slurm)  ## Slurm cluster
plan(batchtools_sge)    ## SGE cluster
```

```
library(future.apply)
plan(future.batchtools::batchtools_sge, workers = 3)

info <- future_lapply(seq_len(nbrOfWorkers()), function(idx) {
  data.frame(idx = idx, hostname = Sys.info()[["nodename"]], pid = Sys.getpid())
})
```

If we peek at the job scheduler queue right after calling `future_lapply()`, we would see something like:

```
$ qstat
job-ID prior name            user  state time
-----------------------------------------------------------
279873 0.000 future_lapply_1 hb    qw    04/26/2023 22:46:54
279889 0.000 future_lapply_2 hb    qw    04/26/2023 22:47:02
279912 0.000 future_lapply_3 hb    qw    04/26/2023 22:47:13
```

These three jobs represent the three futures we created. When completed, we will see something like:

```
info <- do.call(rbind, info)
info
```

```
  idx hostname   pid
1   1 qb3-as92 14596
2   2 qb3-as17 48397
3   3 qb3-as04  8698
```

5.5 Alternatives

The **future.callr** package parallelizes on the local machine using the **callr** package. It works similarly to `multisession`, but can use more than 125 parallel workers[1]:

```
library(future.callr)
plan(callr)
plan(callr, workers = availableCores())  ## default
plan(callr, workers = 2)
```

[1]There is no variable `f` created; instead it is hidden away using the name `...future.v`.

6 The Future API

Previously, we saw several examples of how to use *future assignments* (`%<-%`), e.g.

```
y1 %<-% slow_sum(1:10)
y2 %<-% slow_sum(11:20)
```

```
y1
```

```
[1] 55
```

```
y2
```

```
[1] 155
```

for performing multiple tasks concurrently. That `%<-%` assignment operator doing a lot of things under the hood. A more explicit way of implementing this would be to use the `future()` and `value()` functions, as in:

```
f1 <- future(slow_sum(1:10))
f2 <- future(slow_sum(11:20))
```

```
y1 <- value(f1)
y1
```

```
[1] 55
```

```
y2 <- value(f2)
y2
```

```
[1] 155
```

The `future()` and `value()` are two of three core functions part of the *Future API*, which we will go into details next.

6.1 Three atomic building blocks

There are *three atomic building blocks* part of the Future API that do everything we need for performing tasks concurrently:

- `f <- future(expr)` : evaluates an expression via a future (non-blocking, if possible)
- `r <- resolved(f)` : TRUE if future is resolved, otherwise FALSE (non-blocking)
- `v <- value(f)` : the value of the future expression expr (blocking until resolved)

where `expr` is an R expression. Here are three examples:

```
f <- future(1 + 2)

f <- future(slow_sum(1:10))

f <- future({
  x <- rnorm(10)
  sum(x)
})
```

6.1.1 Mental model: The Future API decouples a regular R assignment into two parts

Let's consider a regular assignment in R:

```
v <- expr
```

To use, this assignment is single operator, but internally it's done in two steps:

1. R evaluates the expression `expr` on the right-hand side (RHS), and
2. assigns the resulting value to the variable `v` on the left-hand side (LHS).

We can think of the Future API as decoupling these two steps and giving us full access to them:

```
f <- future(expr)
v <- value(f)
```

This decoupling is the key feature of all parallel processing!

Let's break this down using our example very slow, implementation of `sum()`;

```
slow_sum <- function(x) {
  sum <- 0

  for (value in x) {
    Sys.sleep(1.0)  ## one-second slowdown per value
    sum <- sum + value
  }

  sum
}
```

For example, if we call:

```
x <- 1:10
v <- slow_sum(x)
v
```

```
[1] 55
```

it takes ten seconds to complete.

We can evaluate this via a future running in the background as:

```
library(future)
plan(multisession) # evaluate futures in parallel

x <- 1:10
f <- future(slow_sum(x))
v <- value(f)
```

When we call:

```
f <- future(slow_sum(x))
```

then:

1. a future is created, comprising:
    - the R expression `slow_sum(x)`,
    - function `slow_sum()`, and
    - integer vector `x`
2. These future components are sent to a parallel worker, which starts evaluating the R expression
3. The `future()` function returns immediately a reference `f` to the future, and before the future evaluation is completed

When we call:

```
v <- value(f)
```

then:

1. the future asks the worker if it's ready or not (using `resolved()` internally)
2. if it is not ready, then it waits until it's ready (blocking)
3. when ready, the results are collected from the worker
4. the value of the expression is returned

As we saw before, there is nothing preventing us from doing other things in-between creating the future and asking for its value, e.g.

```
x <- 1:10

## Create future
f <- future(slow_sum(x))

## We are free to do whatever we want while future is running, e.g.
z <- sd(x)

## Wait for future to be done
v <- value(f)
```

6.1.2 Keep doing other things while waiting

We can use the `resolved()` function to check whether the future is resolved or not. If not, we can choose to do other things, e.g. output a message:

```
f <- future(slow_sum(x))

while (!resolved(f)) {
  message("Waiting ...")
  Sys.sleep(1.0)
}
```

```
Waiting ...
Waiting ...
Waiting ...
Waiting ...
Waiting ...
Waiting ...
Waiting ...
Waiting ...
```

```
message("Done!")
```

```
Done!
```

```
v <- value(f)
v
```

```
[1] 55
```

We can of course do other things than outputting messages, e.g. calculations and checking in on other futures.

6.1.3 Evaluate several things in parallel

There's nothing preventing us from launching more than one future in the background. For example, we can split the summation of `x` into two parts, calculate the sum of each part, and then combine the results at the end:

```r
x_head <- head(x, 5)
x_tail <- tail(x, 5)

v1 <- slow_sum(x_head)          ## ~5 secs (blocking)
v2 <- slow_sum(x_tail)          ## ~5 secs (blocking)
v <- v1 + v2
v
```

```
[1] 55
```

We can do the same in parallel:

```r
f1 <- future(slow_sum(x_head)) ## ~5 secs (in parallel)
f2 <- future(slow_sum(x_tail)) ## ~5 secs (in parallel)

## Do other things
z <- sd(x)

v <- value(f1) + value(f2)     ## ready after ~5 secs
v
```

```
[1] 55
```

We can launch as manual parallel futures as we have parallel workers, e.g.

```r
plan(multisession, workers = 8)
nbrOfWorkers()
```

```
[1] 8
```

```r
plan(multisession, workers = 2)
nbrOfWorkers()
```

```
[1] 2
```

If we launch more than this, then the call to `future()` will block until one of the workers are free again. For example,

```
plan(multisession, workers = 2)
nbrOfWorkers()
```

```
[1] 2
```

```
f1 <- future(slow_sum(x_head))
f2 <- future(slow_sum(x_tail))
```

Immediately after these lines have been be completed by R, both these futures are still *unresolved*:

```
resolved(f1)
```

```
[1] TRUE
```

```
resolved(f2)
```

```
[1] FALSE
```

This is because they need about 5 seconds to complete. If we try to launch another future at this point;

```
f3 <- future(slow_sum(rev(x)))   ## <= blocks here
```

it will *not* return instantly, because it has to wait for one of the parallel workers to be available, i.e. that either of `f1` and `f2`, or both, are resolved.

Immediately after `f3` is created, we will see which it was:

```
resolved(f1)
```

```
[1] TRUE
```

```
resolved(f2)
```

```
[1] FALSE
```

```
resolved(f3)
```

```
[1] FALSE
```

If we then call:

```
value(f1) + value(f2)
```

```
[1] 55
```

```
value(f3)
```

```
[1] 55
```

6.2 Choosing a parallel backend

Above I showed how the *developer* can use `future()`, `value()` and sometimes `resolved()` to implement tasks in parallel.

There was also the `plan()` function, which controls how and where these tasks are processed:

- `plan()` - set how and where futures are evaluated

We should the control of this to the *end user*.

Next, let's look at a few different *parallel backends* we can set.

6.2.1 sequential (default)

```
plan(sequential)

f1 <- future(slow_sum(x_head)) # blocks until done
f2 <- future(slow_sum(x_tail)) # blocks until done

v <- value(f1) + value(f2)
```

6.2.2 multisession: in parallel on local computer

```
plan(multisession, workers = 2)

f1 <- future(slow_sum(x_head)) # in the background
f2 <- future(slow_sum(x_tail)) # in the background

v <- value(f1) + value(f2)
```

What's happening under the hood is:

```
workers <- parallelly::makeClusterPSOCK(2)
plan(cluster, workers = workers)
```

which is very similar to:

```
workers <- parallel::makeCluster(2)
plan(cluster, workers = workers)
```

is you happened to have used the **parallel** package before.

6.2.3 cluster: in parallel on multiple computers

If we have SSH access to other machines with R installed, we can do:

```
hostnames <- c("pi", "remote.server.org")
plan(cluster, workers = hostnames)

f1 <- future(slow_sum(x_head)) # on either 'pi' or 'remote.server.org'
f2 <- future(slow_sum(x_tail)) # on either 'pi' or 'remote.server.org'

v <- value(f1) + value(f2)
```

What's happening under the hood is:

```
hostnames <- c("pi", "remote.server.org")
workers <- parallelly::makeClusterPSOCK(hostnames)
plan(cluster, workers = workers)
```

where `makeClusterPSOCK()` connects to the different machines over SSH using pre-configured SSH keys and reverse tunneling of ports.

The **parallelly** package is a utility package, part of the Futureverse.

6.2.4 There are other parallel backends and more to come

6.2.4.1 future.callr - parallelize locally using callr

The **callr** package can evaluate R expressions in the background on your local computer. The **future.callr** implements a future backend on top of **callr**, e.g.

```
plan(future.callr::callr, workers = 4)
```

This works similarly to `plan(multisession, workers = 4)`, but has the benefit of being able to run more than 125 background workers, which is a limitation of R itself.

6.2.4.2 future.batchtools - parallelize using batchtools

The **batchtools** package is designed to evaluate R expressions via a, so called, job scheduler. Job schedulers are commonly used on high-performance compute (HPC) clusters, where many users run at the same time. The job scheduler allows them to request slots on the system, which often has tens or hundreds of compute nodes. Common job schedulers are Slurm, SGE, and Torque.

The **future.batchtools** implements a future backend on top of **batchtools**.

For example, if you are a user of the Sherlock HPC cluster at Stanford University, you can use:

```
plan(future.batchtools::batchtools_slurm)
```

This will cause `future()` to be submitted to the Slurm job-scheduler queue. When a slot is available, the job is processed on one of the many compute nodes, and when done, the results are stored to file.

Calling `resolved()` will query Slurm whether the job is completed or not. Internally, the `batchtools_slurm` backend calls the `squeue` command to check if the job is done or not.

Calling `value()` will read the results back into R. The `batchtools_slurm` backend relies on the file system for this. At the end of each job, the future framework saves the results to file, which then `value()` reads back.

This future backend has a greater latency, because everything has to be queued on a shared job queue and data and results are communicated via the file system. This backend is useful for long running futures and for the huge throughput that an HPC environment can provide.

6.3 Revisiting the future assignment operator (%<-%)

When we do:

```
v %<-% expr
```

the following is done under the hood[1]:

```
f <- future(expr)
delayedAssign("v", value(f))
```

where `delayedAssign(name, value)` is a function part of R that assigned the value `value` to variable `name`, but not until the variable is used. In our case, this means that:

```
value(f)
```

is only called if, and only if, we "touch" variable `v`. This is why:

```
v %<-% expr
```

can create a future without blocking.

[1]There is no variable `f` created; instead it is hidden away using the name `...future.v`.

7 Non-exportable objects

We already know that some algorithms are such that they cannot be parallelized, e.g. the next iteration depends on the result of the previous iterations.

For example, assume we set up a database connection to a MariaDB database called `sakila` hosted on a remote server using **DBI**:

```
library(DBI)
con <- dbConnect(RMariaDB::MariaDB(),
  host = "relational.fit.cvut.cz", port = 3306,
  dbname = "sakila",
  username = "guest", password = "relational"
)
con
```

```
<MariaDBConnection>
  Host:    relational.fit.cvut.cz
  Server:
  Client:
```

We happen to know there is a table called `film` in this database. We can read in this table into R as a tibble data frame using:

```
film <- dbReadTable(con, "film")
film <- as_tibble(film)
film
```

```
# A tibble: 1,000 x 13
   film_id title       description release_year language_id original_language_id
     <int> <chr>       <chr>              <int>       <int>                 <int>
 1       1 ACADEMY DI~ A Epic Dra~         2006           1                    NA
 2       2 ACE GOLDFI~ A Astoundi~         2006           1                    NA
 3       3 ADAPTATION~ A Astoundi~         2006           1                    NA
 4       4 AFFAIR PRE~ A Fanciful~         2006           1                    NA
```

```
 5       5 AFRICAN EGG A Fast-Pac~         2006           1                    NA
 6       6 AGENT TRUM~ A Intrepid~         2006           1                    NA
 7       7 AIRPLANE S~ A Touching~         2006           1                    NA
 8       8 AIRPORT PO~ A Epic Tal~         2006           1                    NA
 9       9 ALABAMA DE~ A Thoughtf~         2006           1                    NA
10      10 ALADDIN CA~ A Action-P~         2006           1                    NA
# i 990 more rows
# i 7 more variables: rental_duration <int>, rental_rate <dbl>, length <int>,
#   replacement_cost <dbl>, rating <chr>, special_features <chr>,
#   last_update <dttm>
```

7.1 A database connection is only valid in the current R session

Now, say we wish to do this in parallel instead. If we attempt to do:

```
library(future)
plan(multisession)

f <- future({
  df <- dbReadTable(con, "film")
  as_tibble(df)
})
```

It will *not* work;

```
film <- value(f)
```

```
Error: external pointer is not valid
```

The reason is that the database connection (`con`) only works in the R session where it was created. When we tried to use it a parallel worker's R process, it is invalid there. This is certainly not obvious from that error message!

Technically, this has to do with *pointers*, which is a programming term used in low-level programming languages such as C and C++. In this case, we can inspect `con` to see that it indeed has an external pointer:

```
str(con)
```

```
Formal class 'MariaDBConnection' [package "RMariaDB"] with 7 slots
  ..@ ptr                   :<externalptr>
  ..@ host                  : chr "relational.fit.cvut.cz"
  ..@ db                    : chr "sakila"
  ..@ load_data_local_infile: logi FALSE
  ..@ bigint                : chr "integer64"
  ..@ timezone              : chr "UTC"
  ..@ timezone_out          : chr "UTC"
```

If we dig deeper,

```
con@ptr
```

```
<pointer: 0x55bb80fb9bf0>
```

it reveals that its a pointer to a specific address in memory, which exactly how they are used in C and C++. Because it is a memory pointer, that pieces of memory does not exist in the parallel worker. The good thing is that R detects when we send over an object with an external pointer (here `con`). When it detects that, it invalidates the pointer by setting it to null (memory address zero) when sending it over. We can see this is we do:

```
f <- future(con@ptr)
ptr <- value(f)
ptr
```

```
<pointer: (nil)>
```

So, when we try `con` in a parallel workers, the external pointer `con@ptr` is no longer useful. **DBI** detects this invalid pointer when we call `dbReadTable(con, "film")` and throws the error.

7.2 Same problem when saving to file

Note that you have the exact same problem if you would try to save the database connection to file,

```
saveRDS(con, "db_con.rds")
```

and then load it back in again:

```
con2 <- readRDS("db_con.rds")
```

The `con2` object represents a non-working database connection:

```
film <- dbReadTable(con2, "film")
```

```
Error: external pointer is not valid
```

This makes sense, because in the end of the day, it is a connection to a remote database that involves a live connection over internet with authentication, and more. Being able to save its state to file, would be a lot to ask for of the **DBI** package, but also of the remote database server.

7.3 Workaround

A workaround is to create a new database connection in the new R session, or in the parallel worker;

```
library(DBI)
library(future)
plan(multisession)

f <- future({
  con <- dbConnect(RMariaDB::MariaDB(),
    host = "relational.fit.cvut.cz", port = 3306,
    dbname = "sakila",
    username = "guest", password = "relational"
  )

  df <- dbReadTable(con, "film")
  as_tibble(df)
})

film <- value(f)
film
```

```
# A tibble: 1,000 x 13
   film_id title       description release_year language_id original_language_id
```

```
     <int> <chr>       <chr>              <int>       <int>                 <int>
 1       1 ACADEMY DI~ A Epic Dra~         2006           1                    NA
 2       2 ACE GOLDFI~ A Astoundi~         2006           1                    NA
 3       3 ADAPTATION~ A Astoundi~         2006           1                    NA
 4       4 AFFAIR PRE~ A Fanciful~         2006           1                    NA
 5       5 AFRICAN EGG A Fast-Pac~         2006           1                    NA
 6       6 AGENT TRUM~ A Intrepid~         2006           1                    NA
 7       7 AIRPLANE S~ A Touching~         2006           1                    NA
 8       8 AIRPORT PO~ A Epic Tal~         2006           1                    NA
 9       9 ALABAMA DE~ A Thoughtf~         2006           1                    NA
10      10 ALADDIN CA~ A Action-P~         2006           1                    NA
# i 990 more rows
# i 7 more variables: rental_duration <int>, rental_rate <dbl>, length <int>,
#   replacement_cost <dbl>, rating <chr>, special_features <chr>,
#   last_update <dttm>
```

7.4 Futureverse can help us detect this before it happens

The problem of *non-exportable objects* it not just for database connections. It happens for a large number of other classes of objects. Most of them have one thing in common: the hold a "reference" to some external resources, e.g. a file connection, a website connection, a database connection, a handler to an in-memory object living in a Python or a Java process running in the background. However, there are also cases where the references is an external pointer to a piece of the memory on the current machine.

For a list of known cases, see https://future.futureverse.org/articles/future-4-non-exportable-objects.html.

The **future** package can scan for external pointers, and other types of "references". We can use this to help us protect against these type of mistakes:

```
library(future)
options(future.globals.onReference = "error")                    ①
plan(multisession)

f <- future({
  df <- dbReadTable(con, "film")
  as_tibble(df)
})
```

① R option to tell **future** to prevent objects with external pointers from being exported to a parallel worker.

```
Error: Detected a non-exportable reference ('externalptr') in one of the globals ('con' of cl

Timing stopped at: 0.001 0 0.002
```

This is not the default setting, because there exist objects with external points that can indeed be exported. For example, `data.table` objects have external pointers, but the **data.table** package is clever enough to ignore it, if the pointer is invalid. For example,

```
library(data.table)

dt <- as.data.table(iris)
dt
```

```
     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
 ---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

We can see the external pointer, if we use:

```
str(dt)
```

```
Classes 'data.table' and 'data.frame':  150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

The pointer is:

```
attr(dt, ".internal.selfref")
```

```
<pointer: 0x55bb7aad4dd0>
```

If we would set the above R option, we would get an error if we would try to send `dt` to a parallel worker. We don't want that, because it works:

```
f <- future(summary(dt))
value(f)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

If we inspect the pointer of `dt` on the parallel worker, we'll find that it is null (as expected);

```
f <- future(attr(dt, ".internal.selfref"))
ptr <- value(f)
ptr
```

```
<pointer: (nil)>
```

If we round trip to the file system, we see this familiar behavior of external pointers being set to null by R:

```
saveRDS(dt, "dt.rds")
attr(dt, ".internal.selfref")
```

```
<pointer: 0x55bb7aad4dd0>
```

```
dt2 <- readRDS("dt.rds")
attr(dt2, ".internal.selfref")
```

```
<pointer: (nil)>
```

7.5 What about forked parallelization?

One may think that *forked* parallel processing could be a workaround. When using *forks*, the operating system will "clone" our main R session and perfectly replicate everything in the child parallel process.

Let's try with our database example;

```
library(DBI)
con <- dbConnect(RMariaDB::MariaDB(),
  host = "relational.fit.cvut.cz", port = 3306,
  dbname = "sakila",
  username = "guest", password = "relational"
)
con
```

```
<MariaDBConnection>
  Host:    relational.fit.cvut.cz
  Server:
  Client:
```

```
library(future)
plan(multicore)   ## forked parallelization

f <- future({
  df <- dbReadTable(con, "film")
  as_tibble(df)
})

film <- value(f)
film
```

```
# A tibble: 1,000 x 13
   film_id title       description release_year language_id original_language_id
     <int> <chr>       <chr>              <int>       <int>                 <int>
 1       1 ACADEMY DI~ A Epic Dra~         2006           1                    NA
 2       2 ACE GOLDFI~ A Astoundi~         2006           1                    NA
 3       3 ADAPTATION~ A Astoundi~         2006           1                    NA
 4       4 AFFAIR PRE~ A Fanciful~         2006           1                    NA
 5       5 AFRICAN EGG A Fast-Pac~         2006           1                    NA
 6       6 AGENT TRUM~ A Intrepid~         2006           1                    NA
 7       7 AIRPLANE S~ A Touching~         2006           1                    NA
 8       8 AIRPORT PO~ A Epic Tal~         2006           1                    NA
 9       9 ALABAMA DE~ A Thoughtf~         2006           1                    NA
10      10 ALADDIN CA~ A Action-P~         2006           1                    NA
# i 990 more rows
# i 7 more variables: rental_duration <int>, rental_rate <dbl>, length <int>,
#   replacement_cost <dbl>, rating <chr>, special_features <chr>,
#   last_update <dttm>
```

It certainly looks like it worked! However, while doing this, we managed to confuse **DBI** and MariaDB. If we try to use `con` again, we get:

```
film <- dbReadTable(con, "film")
```

```
Error: Lost connection to MySQL server during query [2013]
```

Conclusion, it is tempting to think *forked* processing can solve things that other parallelization backends cannot handle, but it is often the devil in disguise.

8 foreach() is not a for-loop

For-loops are special in the way they *can assign values to objects outside* of the for-loop. For example,

```
xs <- list()
ys <- list()
last_idx <- 0
for (idx in 1:3) {
  xs[[idx]] <- letters[idx]
  ys[[idx]] <- LETTERS[idx]
  last_idx <- idx
}
```

assigns to both `xs` and `ys`. We also see that `last_idx` is updated in every iteration, and, when the for loop completes, it holds:

```
last_idx
```

```
[1] 3
```

In contrast, we *cannot* do the same for map-reduce calls, such as `lapply()`, because they *return* results, but *cannot assign outside*.

8.1 Super assignment (<<-) is not a solution

Warning, using "super" assignments (`<<-`), as in:

```
xs <- list()
ys <- list()
last_idx <- 0
void <- lapply(1:3, function(idx) {
  xs[[idx]] <<- letters[idx]
  ys[[idx]] <<- LETTERS[idx]
```

```
  last_idx <<- idx
})
```

or, similarly, `assign(..., envir = parent.frame())`, is considered a bad practise for many reasons. **Please, do *not* use such hacks!** (they will come and bite you if you try - trust me).

Previously, I said that any `lapply()` call can be replaced with a `future_lapply()` such that it can run in parallel. What would happen if we would go ahead and use the above `<<-` hack? Let us try:

```
library(future.apply)
plan(multisession)

xs <- list()
ys <- list()
last_idx <- 0
void <- future_lapply(1:3, function(idx) {
  xs[[idx]] <<- letters[idx]
  ys[[idx]] <<- LETTERS[idx]
  last_idx <<- idx
})
```

If we check `xs`, `ys`, and `last_idx` afterward;

```
str(xs)
```

```
list()
```

```
str(ys)
```

```
list()
```

```
last_idx
```

```
[1] 0
```

we find that they are empty and zero.

Q. Why is that?

The reason is that the expressions:

```
xs[[idx]] <<- letters[idx]
ys[[idx]] <<- LETTERS[idx]
last_idx <<- idx
```

are evaluated in another R process. The assignment to `xs`, `ys`, and `last_idx` is done to the global environment of that R process, which is *not* the same as the global environment of our main R session. In our main R session, the only assignment to `xs` and `ys` was from our initial:

```
xs <- list()
ys <- list()
last_idx <- 0
```

assignments, which is why they are still the same.

Now, assume for a moment it would indeed be possible to use `<<-` to assign to the main R session also from parallel processes. If so, what value should `last_idx` have at the very end? That would depend on in which order the parallel tasks would complete. For instance, imagine the first iteration (`idx = 1`) would be very slow and therefore finish last. Would you then expect `last_idx` to be `1` or `3`?

Conclusion: It is not possible, and it does not make sense, to assign to the global environment when running in parallel!

8.2 Return instead of assign in map-reduce calls

The solution for map-reduce functions, such as `lapply()`, is to return all results and split afterward, e.g.

```
res <- lapply(1:3, function(idx) {
  data.frame(x = letters[idx], y = LETTERS[idx], idx = idx)
})
xs <- lapply(res, `[[`, "x")
ys <- lapply(res, `[[`, "y")
last_idx <- res[[length(res)]][["last_idx"]]
rm(res)
```

```
  str(xs)
```

```
List of 3
 $ : chr "a"
 $ : chr "b"
 $ : chr "c"
```

```
  str(ys)
```

```
List of 3
 $ : chr "A"
 $ : chr "B"
 $ : chr "C"
```

```
  last_idx
```

```
NULL
```

This strategy works in parallel too:

```
  library(future.apply)
  plan(multisession)

  res <- future_lapply(1:3, function(idx) {
    list(x = letters[idx], y = LETTERS[idx], idx = idx)
  })
  xs <- lapply(res, `[[`, "x")
  ys <- lapply(res, `[[`, "y")
  last_idx <- res[[length(res)]][["idx"]]
  rm(res)

  str(xs)
```

```
List of 3
 $ : chr "a"
 $ : chr "b"
 $ : chr "c"
```

```
str(ys)
```

```
List of 3
 $ : chr "A"
 $ : chr "B"
 $ : chr "C"
```

```
last_idx
```

```
[1] 3
```

8.3 foreach() is a map-reduce function

The main thing to understand is that `foreach()` does *not* work like a for-loop. If you would try, say

```
library(doFuture)
registerDoFuture()
plan(multisession)

xs <- list()
ys <- list()
last_idx <- 0
void <- foreach(idx = 1:3, .export = c("xs", "ys")) %dopar% {
  xs[[idx]] <- letters[idx]
  ys[[idx]] <- LETTERS[idx]
  last_idx <- idx
}
```

you'll find that:

```
str(xs)
```

```
list()
```

```
str(ys)
```

```
list()
```

```
last_idx
```

```
[1] 0
```

This is because `foreach()` is a map-reduce function. It is only its name and the `%dopar%` operator that makes it *visually* resemble a for-loop although it isn't one. To further clarify this, if it would not be for the `%dopar%` operator, the original creator would probably have designed `foreach()` to take a function just `lapply()`, e.g.

```
void <- foreach(idx = 1:3, function(idx) {
  ...
})
```

If that would have been the case, it would be clear that `foreach()` is just another map-reduce function lust like `lapply()` and `map()` of the **purrr** package.

To conclude, we should always use `foreach()` as a map-reduce function, e.g.

```
library(doFuture)
plan(multisession)

res <- foreach(idx = 1:3) %dofuture% {
  list(x = letters[idx], y = LETTERS[idx], idx = idx)
}
xs <- lapply(res, `[[`, "x")
ys <- lapply(res, `[[`, "y")
last_idx <- res[[length(res)]][["idx"]]
rm(res)

str(xs)
```

```
List of 3
 $ : chr "a"
 $ : chr "b"
 $ : chr "c"
```

```
str(ys)
```

```
List of 3
 $ : chr "A"
 $ : chr "B"
 $ : chr "C"
```

```
last_idx
```

```
[1] 3
```

9 mclapply() - is it really magic?

9.1 Output is at best fake from mclapply()

```
library(parallel)

y <- mclapply(1:3, print)
```

The output produced by `print(1)`, `print(2)`, and `print(3)` on the parallel workers, may or may not be visible. If you call the above in in the RStudio Console, or in a RMarkdown document, there will be no output visible. If you run R in a Linux terminal, you will probably see something like:

```
[1] 1
[1] 3
[1] 2
```

That is more luck than skill by R - it is the Linux terminal that saves us by relaying the output. Note also that the output is in whatever parallel worker calls `print()` first. There is also a risk that the different output interweave each other, e.g.

```
[1[1]
] 1
 3
[1] 2
```

We can confirm that the output never reaches the main R session by testing with `capture.output()`. If we do this using a regular `lapply()` call, then we get:

```
output <- capture.output(y <- lapply(1:3, print))
output
```

```
[1] "[1] 1" "[1] 2" "[1] 3"
```

However, when we use `mclapply()`, we get nothing:

```
output <- capture.output(y <- mclapply(1:3, print))
output
```

```
character(0)
```

This is actually not that surprising. `capture.output()` sets up a *sink*, which internally writes output to a `textConnection()` connection. When used in a forked parallel workers, this connection ends up writing to its locally cloned text connection. That captured output is *not* available in the parent R session. In other words, all "captured" output happening in parallel workers are lost.

In contrast, all functions in the Futureverse relays output in parallel workers to the main R session. It is also done such that the "natural" order is respected. For example,

```
library(future.apply)
plan(multicore)                                                    ①

y <- future_lapply(1:3, print)
```

① `multicore` uses *forked* parallelization based on the same code as `mclapply()`.

```
[1] 1
[1] 2
[1] 3
```

and

```
output <- capture.output(y <- future_lapply(1:3, print))
output
```

```
[1] "[1] 1" "[1] 2" "[1] 3"
```

9.2 Warnings are lost by mclapply()

What happens with warning? Consider:

```
y <- lapply(-1:1, sqrt)
```

```
Warning in FUN(X[[i]], ...): NaNs produced
```

which produces a warning from `sqrt(-1)`. If we try the same with `mclapply()`, that *warning is lost*:

```
library(parallel)

y <- mclapply(-1:1, sqrt)
```

In contrast, all functions in the Futureverse relays warnings, and any other type of *condition*, in parallel workers to the main R session. It is also done such that the "natural" order is respected. For example,

```
library(future.apply)
plan(multicore)

y <- future_lapply(-1:1, sqrt)
```

```
Warning in ...future.FUN(...future.X_jj, ...): NaNs produced
```

9.3 Errors are mangled

What happens with errors? Consider:

```
y <- lapply(list(1, 2, "a"), sqrt)
```

```
Error in FUN(X[[i]], ...): non-numeric argument to mathematical function
```

which produces an error because of `sqrt("a")`. If we try the same with `mclapply()`, that *error is turned into an obscure warning*:

```
library(parallel)

y <- mclapply(list(1, 2, "a"), sqrt)
```

```
Warning in mclapply(list(1, 2, "a"), sqrt): scheduled core 1 encountered error
in user code, all values of the job will be affected
```

Because it is just a warning, it means that your code keeps running as nothing really happened!

This is one example how mistakes in scientific pipelines can go by unnoticed. If we inspect `y`, we can see there is information about the error:

```
str(y)
```

```
List of 3
 $ : 'try-error' chr "Error in FUN(X[[i]], ...) : non-numeric argument to mathematical functi
  ..- attr(*, "condition")=List of 2
  .. ..$ message: chr "non-numeric argument to mathematical function"
  .. ..$ call   : language FUN(X[[i]], ...)
  .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
 $ : num 1.41
 $ : 'try-error' chr "Error in FUN(X[[i]], ...) : non-numeric argument to mathematical functi
  ..- attr(*, "condition")=List of 2
  .. ..$ message: chr "non-numeric argument to mathematical function"
  .. ..$ call   : language FUN(X[[i]], ...)
  .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

To make sure that errors are not slipping by unnoticed, we need to do something like:

```
is_error <- vapply(y, inherits, "try-error", FUN.VALUE = NA)
if (any(is_error)) {
  ## error objects are stored in attributes
  first_error <- attr(y[is_error][[1]], "condition")
  stop("Detected one or more errors: ", conditionMessage(first_error))
}
```

```
Error in eval(expr, envir, enclos): Detected one or more errors: non-numeric argument to mathl
```

In contrast, all functions in the Futureverse relays errors. For example,

```
library(future.apply)
plan(multicore)
```

```
y <- future_lapply(list(1, 2, "a"), sqrt)
```

```
Error in ...future.FUN(...future.X_jj, ...): non-numeric argument to mathematical function
```

9.4 What happens when a parallel crashes?

Sometime a parallel workers crashes. This can happen if there are too many processes running on the same machine and all memory gets consumed. Then the operating systems, particularly on Linux, decides to kill process in order for the machine not to go down. Another reason could be that the R code calls some incorrect C code, and it terminates with a segfault because of that, e.g.

```
 *** caught illegal operation ***
address 0x2b3a8b234ccd, cause 'illegal operand'
```

We can emulate terminating the current R process by calling `tools::pskill(Sys.getpid())`. For example,

```
y <- lapply(1:3, function(idx) {
  if (idx == 2) tools::pskill(Sys.getpid())
  idx
})
```

This will result in R terminating abruptly:

```
Terminated
```

Now, what happens if we terminate a parallel worker? If we use `mclapply()`, this is what happens:

```
library(parallel)

y <- mclapply(1:3, function(idx) {
  if (idx == 2) tools::pskill(Sys.getpid())
  idx
})
```

```
Warning in mclapply(1:3, function(idx) {: scheduled core 2 did not deliver a
result, all values of the job will be affected
```

Again, just a warning! Danger!

In contrast, the Futureverse detects when a parallel workers crashes and gives an informative error message:

```
library(future.apply)
plan(multicore)

y <- future_lapply(1:3, function(idx) {
  if (idx == 2) tools::pskill(Sys.getpid())
  idx
})
```

```
Warning in mccollect(jobs = jobs, wait = TRUE): 1 parallel job did not deliver
a result
```

```
Error: Failed to retrieve the result of MulticoreFuture (future_lapply-2) from the forked wo
```

Summary

In these two lectures, we've learned that:

- Parallelization does not have to be hard
- Futureverse simplifies parallelization in R (disclaimer!)
- `foreach()` is, lust like `lapply()` *not* a for-loop
- There are things we *cannot* parallelize
- *Forked* parallel processing is neat, but should be used with caution

Much more information can be found at https://www.futureverse.org.

Part III

Appendix

References

Bengtsson, Henrik. 2021. "A Unifying Framework for Parallel and Distributed Processing in r Using Futures." *The R Journal* 13 (2): 208–27. https://doi.org/10.32614/RJ-2021-048.