# Interactive debugging

First out among the new features, and a long-running feature request, is the addition of argument `split` to `plan()`, which allows us to split, or "tee", any output produced by futures.

The default is `split = FALSE` for which standard output and conditions are captured by the future and only relayed after the future has been resolved, i.e. the captured output is displayed and re-signaled on the main R session when value of the future is queried. This emulates what we experience in R when not using futures, e.g. we can add temporary `print()` and `message()` statements to our code for quick troubleshooting. You can read more about this in blog post 'future 1.9.0 – Output from The Future'.

However, if we want to use `debug()` or `browser()` for interactive debugging, we quickly realize they're not very useful because no output is visible, which is because also their output is captured by the future. This is where the new "split" feature comes to rescue. By using `split = TRUE`, the standard output and all non-error conditions are split ("tee:d") on the worker's end, while still being captured by the future to be relayed back to the main R session at a later time. This means that we can debug 'sequential' future interactively. Here is an illustration of using `browser()` for debugging a future:

```
> library(future)
> plan(sequential, split = TRUE)
> mysqrt <- function(x) { browser(); y <- sqrt(x); y }
> f <- future(mysqrt(1:3))
Called from: mysqrt(1:3)
Browse[1]> str(x)
 int [1:3] 1 2 3
Browse[1]>
debug at #1: y <- sqrt(x)
Browse[2]>
debug at #1: y
Browse[2]> str(y)
 num [1:3] 1 1.41 1.73
Browse[2]> y[1] <- 0
Browse[2]> cont

> v <- value(f)
Called from: mysqrt(1:3)
 int [1:3] 1 2 3
debug at #1: y <- sqrt(x)
debug at #1: y
 num [1:3] 1 1.41 1.73

> v
[1] 0.000000 1.414214 1.732051
```

*Comment*: Note how the output produced while debugging is relayed also when `value()` is called. This is a somewhat unfortunate side effect from futures capturing *all* output produced while they are active.

# Preserved logging on workers (e.g. future.batchtools)

The added support for `split = TRUE` also means that we can now preserve all output in any log files that might be produced on parallel workers. For example, if you use **future.batchtools** on a Slurm scheduler, you can use `plan(future.batchtools::batchtools_slurm, split = TRUE)` to make sure standard output, messages, warnings, etc. are ending up in the **batchtools** log files while still being relayed to the main R session at the end. This way we can inspect cluster jobs while they still run, among other things. Here is a proof-of-concept example using a 'batchtools_local' future:

```
> library(future.batchtools)
> plan(batchtools_local, split = TRUE)
> f <- future({ message("Hello world"); y <- 42; print(y); sqrt(y) })
> v <- value(f)
[1] 42
Hello world
> v
[1] 6.480741
> loggedOutput(f)
 [1] "### [bt]: This is batchtools v0.9.14"
 [2] "### [bt]: Starting calculation of 1 jobs"
 [3] "### [bt]: Setting working directory to '/home/alice/repositories/
future'"
 [4] "### [bt]: Memory measurement disabled"
 [5] "### [bt]: Starting job [batchtools job.id=1]"
 [6] "### [bt]: Setting seed to 15794 ..."
 [7] "Hello world"
 [8] "[1] 42"
 [9] ""
[10] "### [bt]: Job terminated successfully [batchtools job.id=1]"
[11] "### [bt]: Calculation finished!"
```

Without `split = TRUE`, we would not get lines 7 and 8 in the **batchtools** logs.

# Near-live progress updates also from 'multicore' futures

Second out among the new features is 'multicore' futures, which now join 'sequential', 'multisession', and (local and remote) 'cluster' futures in the ability of relaying progress updates of **progressr** in a near-live fashion. This means that all of our most common parallelization backends support near-live progress updates. If this is the first time you hear of **progressr**, here's an example of how it can be used in parallel processing:

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

library(progressr)
handlers("progress")

xs <- 1:5
with_progress({
```

```
  p <- progressor(along = xs)
  y <- future_lapply(xs, function(x, ...) {
    Sys.sleep(6.0-x)
    p(sprintf("x=%g", x))
    sqrt(x)
  })
})

# [================>------------------------------]  40% x=2
```

Note that the progress updates signaled by `p()`, updates the progress bar almost instantly, even if the parallel workers run on a remote machine.

# Multisession futures agile to changes in R's library path

Third out is 'multisession' futures. It now automatically inherits the package library path from the main R session. For instance, if you use `.libPaths()` to adjust your library path and *then* call `plan(multisession)`, the multisession workers will see the same packages as the parent session. This change is based on a feature request related to RStudio Connect. With this update, it no longer matters which type of local futures you use – 'sequential', 'multisession', or 'multicore' – your future code has access to the same set of installed packages.

As a proof of concept, assume that we add `tempdir()` as a new folder to R's library path;

```
> .libPaths(c(tempdir(), .libPaths()))
> .libPaths()
[1] "/tmp/alice/RtmpwLKdrG"
[2] "/home/alice/R/x86_64-pc-linux-gnu-library/4.0-custom"
[3] "/home/alice/software/R-devel/tags/R-4-0-3/lib/R/library"
```

If we then launch a 'multisession' future, we find that it uses the same library path;

```
> library(future)
> plan(multisession)
> f <- future(.libPaths())
> value(f)
[1] "/tmp/alice/RtmpwLKdrG"
[2] "/home/alice/R/x86_64-pc-linux-gnu-library/4.0-custom"
[3] "/home/alice/software/R-devel/tags/R-4-0-3/lib/R/library"
```

# Best practices for package developers

I've added a vignette 'Best Practices for Package Developers', which hopefully provides some useful guidelines on how to write and validate future code so it will work on as many parallel backends as possible.

# Saying goodbye to 'multiprocess' – but don't worry …

Ok, lets discuss what is being removed. Using `plan(multiprocess)`, which was just an alias for "`plan(multicore)` on Linux and macOS and `plan(multisession)` on MS Windows", is now deprecated. If used, you will get a one-time warning:

```
> plan(multiprocess)
Warning message:
Strategy 'multiprocess' is deprecated in future (>= 1.20.0). Instead,
explicitly
specify either 'multisession' or 'multicore'. In the current R session,
'multiprocess' equals 'multicore'.
```

I recommend that you use `plan(multisession)` as a replacement for `plan(multiprocess)`. If you are on Linux or macOS, and are 100% sure that your code and all its dependencies is fork-safe, then you can also use `plan(multicore)`.

Although 'multiprocess' was neat to use in documentation and examples, it was at the same time ambiguous, and it risked introducing a platform-dependent behavior to those examples. For instance, it could be that the parallel code worked only for users on Linux and macOS because some non-exportable globals were used. If a user or MS Windows tried the same code, they might have gotten run-time errors. Vice versa, it could also be that code works on MS Windows but not on Linux or macOS. Moreover, in **future** 1.13.0 (2019-05-08), support for 'multicore' futures was disabled when running R via RStudio. This was done because forked parallel processing was deemed unstable in RStudio. This meant that a user on macOS who used `plan(multiprocess)` would end up getting 'multicore' futures when running in the terminal while getting 'multisession' futures when running in RStudio. These types of platform-specific, environment-specific user experiences were confusing and complicates troubleshooting and communications, which is why it was decided to move away from 'multiprocess' in favor of explicitly specifying 'multisession' or 'multicore'.

# Saying goodbye to 'local = FALSE' – a good thing

In an effort of refining the Future API, the use of `future(..., local = FALSE)` is now deprecated. The only place where it is still supported, for backward compatible reason, is when using 'cluster' futures that are persistent, i.e. `plan(cluster, ..., persistent = TRUE)`. If you use the latter, I recommended that you start thinking about moving away from using `local = FALSE` also in those cases. Although `persistent = TRUE` is rarely used, I am aware that some of you got use cases that require objects to remain on the parallel workers also after a future has been resolved. If you have such needs, please see future Issue #433, particularly the parts on "sticky globals". Feel free to add your comments and suggestions for how we best could move forward on this. The long-term goals is to get rid of both `local` and `persistent` in order to harmonize the Future API across *all* future backends.

For recent bug fixes, please see the package NEWS.

# What's on the horizon?

There are still lots of things on the roadmap. In no specific order, here are the few things in the works:

- Sticky globals for caching globals on workers. This will decrease the number of globals that need to be exported when launching futures. It addresses several related feature requests, e.g. future Issues #273, #339, #346, #431, and #437.

- Ability to terminate futures (for backends supporting it), which opens up for the possibility of restarting failed futures and more. This is a frequently requested feature, e.g. Issues #93, #188, #205, #213, and #236.

- Optional, zero-cost generic hook function. Having them in place opens up for adding a framework for doing time-and-memory profiling/benchmarking futures and their backends. Being able profile futures and their backends will help identify bottlenecks and improve the performance on some of our parallel backends, e.g. Issues #59, #142, #239, and #437.

- Add support for global calling handlers in **progressr**. This is not specific to the future framework but since its closely related, I figured I mention this here too. A global calling handler for progress updates would remove the need for having to use `with_progress()` when monitoring progress. This would also help resolve the common problem where package developers want to provide progress updates without having to ask the user to use `with_progress()`, e.g. **progressr** Issues #78, #83, and #85.

That's all for now – Happy futuring!