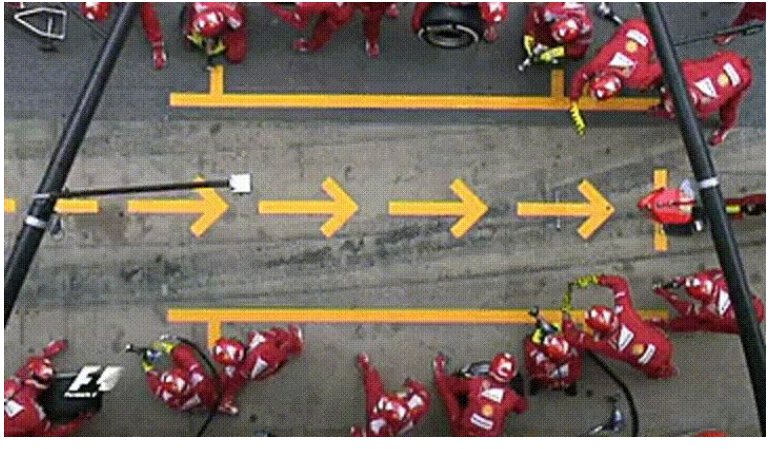
Future: Simple Async, Parallel & Distributed Processing in R

Why and What's New?



Henrik Bengtsson (@HenrikBengtsson)
University of California San Francisco, R Foundation, R Consortium

Parallelization should be simple

```
x <- 1:20
y <- lapply(x, slow)
```

```
      Main R session:
 1m:  y[[1]] <- slow(x[1])
 2m:  y[[2]] <- slow(x[2])

           ...

20m: y[[20]] <- slow(x[20])
```

Time: 20 mins

```
x <- 1:20
y <- mclapply(x, slow, mc.cores=2)
```

```
      Parallel worker #1:    Parallel worker #2:
 1m:  y[[1]] <- slow(x[1])   y[[11]] <- slow(x[11])
 2m:  y[[2]] <- slow(x[2])   y[[12]] <- slow(x[12])
            ...                    ...
10m: y[[10]] <- slow(x[10])  y[[20]] <- slow(x[20])
```

Time: 10 mins

Overwhelming to get started

- So many parallel API - which one should I choose?
  - `mclapply()`, `parLapply()`, `foreach()`, ...
- What operating systems should I support?
  - I use Linux. Will work on Windows and macOS?
- Will it scale?
- Do I need to maintain two code bases - sequential and parallel?
- **`Error in { : task 1 failed - "object 'data' not found"`**

R package: future

Dan LaBar
@embiggenData

- A simple, unifying solution for parallel APIs
- "Write once, run anywhere"
- 100% cross platform
- Easy to install (< 0.5 MiB total)
- Very well tested, lots of CPU mileage, used in production
- Things “just work”

All we need is three building blocks

```r
f <- future(expr)    # evaluate in parallel
r <- resolved(f)     # check if done
v <- value(f)        # wait & get result
```

This was invented in 1975

```r
future_lapply <- function(X, FUN, ...) {
  futures <- lapply(X, function(x) future(FUN(x, ...)))
  lapply(futures, value)
}
```

Stay with your favorite coding style

```
# Base R style (R & future.apply)
y <- lapply(x, slow)
y <- future_lapply(x, slow)

# Tidyverse style (purrr & furrr)  [Hadley W, Davis V]
y <- x %>% map(slow)
y <- x %>% future_map(slow)

# Foreach style (foreach & doFuture)    [Steve Weston]
y <- foreach(z = x) %do% slow(z)
y <- foreach(z = x) %dopar% slow(z)
```

User chooses how to parallelize

- sequential
  `plan(sequential)`
- parallelize on local machine
  `plan(multisession)`
- multiple local or remote computers, or cloud compute services
  `plan(cluster, workers=c("n1", "m2.uni.edu", "vm.cloud.org"))`
- High-performance compute (HPC) cluster
  `plan(batchtools_slurm)`

Your future code remains the same!

Worry-free but does it work?

On CRAN since 2015
Adoptions: **drake**, **shiny** (async), …

Tested on Linux, macOS, Solaris, Windows
Tested on old and new versions of R
Revdep checks on > 100 packages

All **foreach**, **plyr**, **caret**, **glmnet**, ...
`example()`:s validated with all future backends

future.tests - conformance validation of parallel backends
(supported by an R Consortium grant)

What's new?

Output, Warnings, Errors

Output and warnings behave consistently for all parallel backends

```
> x <- c(-1, 10, 30)
> y <- lapply(x, function(z) {
    message("z = ", z)
    log(z)
  })
z = -1
z = 10
z = 30
Warning message:
In FUN(X[[i]], ...) : NaNs produced
>
```

Output and warnings behave consistently for all parallel backends

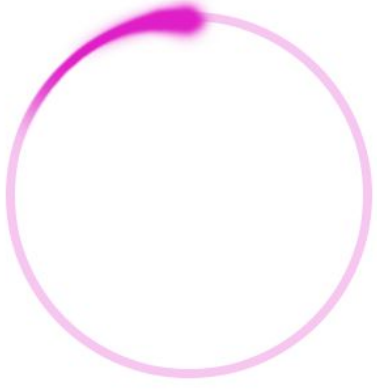
```
> x <- c(-1, 10, 30)
> y <- mclapply(x, function(z) {
    message("z = ", z)
    log(z)
  })
>
```

Output and warnings behave consistently for all parallel backends

```
> x <- c(-1, 10, 30)
> y <- future_lapply(x, function(z) {
    message("z = ", z)
    log(z)
  })
z = -1
z = 10
z = 30
Warning message:
In FUN(X[[i]], ...) : NaNs produced
>
```

What's new?

Progress Updates



progressr - Inclusive, Unifying API for Progress Updates

Works anywhere - including futures, purrr, lapply, foreach, for/while loops, ...

API for Developers:

```
p <- progressor(along=x)
p(msg)
```

<u>Developer decides:</u>

where in the code progress updates should be signaled

API for Users:

```
with_progress({ expr })
```

<u>User decides:</u>

if, when, and how progress updates are presented

Developer focuses on providing updates

Package code

```
snail <- function(x) {
  p <- progressor(along=x)
  y <- sapply(x, function(z) {
    p(paste0("z=", z))
    slow(z)
  }
  sum(y)
}
```

User

```
> x <- 1:50
> with_progress(y <- snail(x))
  [================>--]  90% z=45
```

User decides how progress is presented

```
# without progress updates
> x <- 1:50
> y <- snail(x)
```

```
> handlers("beepr")
> with_progress(y <- snail(x))
```

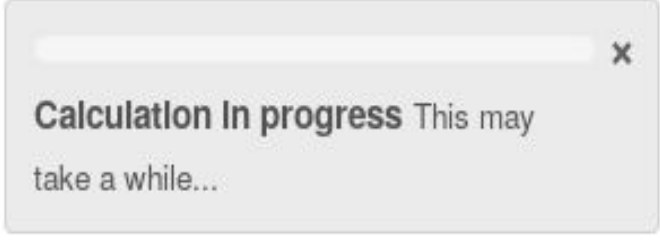
♬ ♪ ♪ ♪ … ♬

```
> handlers("progress", "beepr")
> with_progress(y <- snail(x))
[======>-----------]  40% z=20
```

♬ ♪ ♪ ♪ … ♬

Works also with Shiny

withProgressShiny()



What's new?

future + progressr = ❤️

Now future supports live progress updates

```
snail <- function(x) {
  p <- progressor(along=x)
  y <- future_sapply(x, function(z) {
    p(paste0("z=", z, " by ", Sys.getpid()))
    slow(z)
  })
  sum(y)
}
```

```
> handlers("progress", "beepr")
> plan(multisession)
> with_progress(y <- snail(x))
[=>----------]  10% z=38 by 3001
```

♬ ♪

R 4.0.0:

global calling handlers 🙏

<= with_progress() not needed

Take home: future = worry-free parallelization

- Developer: *what* to parallelize <-> User: *how* to parallelize
- Stay with your favorite coding style
- Automagic, e.g. globals, packages, output, warnings, errors, *progress*

github.com/HenrikBengtsson
@HenrikBengtsson