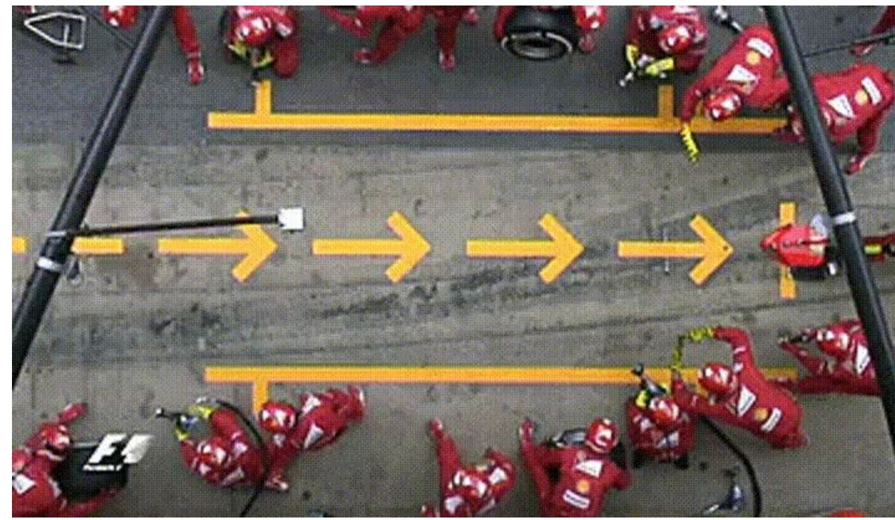
Future: Simple Parallel and Distributed Processing in R

Henrik Bengtsson
University of California
San Francisco



Acknowledgments:

- Organizers, Volunteers, and Sponsors
- R Core, CRAN, devels, and users!
- R Consortium
- Gábor Csárdi

@HenrikBengtsson
HenrikBengtsson
jottr.org

We parallelize software for various reasons

Parallel & distributed processing can be used to:

- speed up processing (wall time)
- lower memory footprint (per machine)
- Other reasons, e.g. asynchronous UI

Concurrency in R

```
X <- list(a=1:50, b=51:100, c=101:150)

y <- list()
y$a <- sum(X$a)
y$b <- sum(X$b)
y$c <- sum(X$c)

y <- list()
for (name in names(X)) {
  y[[name]] <- sum(X[[name]])
}

y <- lapply(X, sum)
```

R comes with built-in parallelization

```
X <- list(a=1:50, b=51:100, c=101:150)
y <- lapply(X, slow_sum)                    # 3 minutes
```

This can be parallelized on Unix & macOS (becomes non-parallel on Windows) as:

```
library(parallel)
y <- mclapply(X, slow_sum, mc.cores=3)    # 1 minute
```

To parallelize also on Windows, we can do:

```
library(parallel)
workers <- makeCluster(3)
clusterExport(workers, "slow_sum")
y <- parLapply(X, slow_sum, cl=workers)  # 1 minute
```

PROBLEM: Different APIs for different parallelization strategies

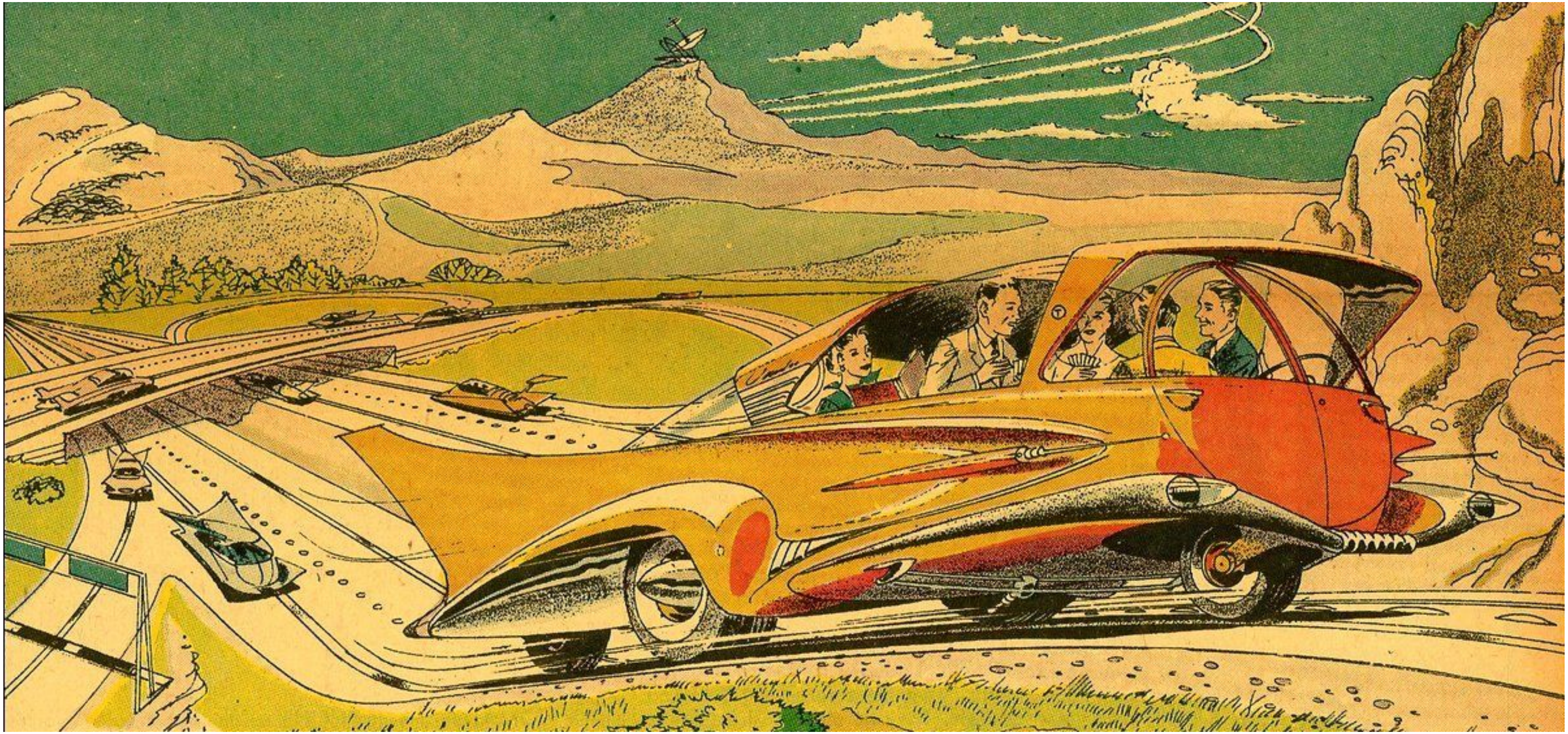
Developer:

- Which parallel API should I use?
- What operating systems are users running?
- I don’t have Windows; can’t be bothered
- \- Hmm… It should work?!?
  \- Oh, I forgot to test on macOS.

User:

- I wish this awesome package could parallelize on Windows :(
- \- Weird, others say it works for them but for me it doesn't!?

Welcome to the Future



R package: future

CRAN 1.14.0

future	
parallel	**globals**

- "Write once, run anywhere"
- 100% cross platform
- A simple unified API
- Easy to install (< 0.5 MiB total)
- Very well tested, lots of CPU mileage

Other key strengths:

- automatically exports **global variables**
- automatically relays:
  - **stdout**
  - **conditions**, e.g. messages and warnings
- works with any type of parallel backends

A Future is ...

- A future is an abstraction for a value that will be available later
- The value is the result of an evaluated expression
- The state of a future is either unresolved or resolved

An R assignment:

```
v <- expr
```

Future API:

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

Friedman & Wise (1976, 1977), Hibbard (1976), Baker & Hewitt (1977)

Example: Sum of 1:100

```
> slow_sum(1:100)             # 2 minutes
[1] 5050

> a <- slow_sum(1:50)         # 1 minute
> b <- slow_sum(51:100)       # 1 minute
> a + b
[1] 5050
```

Example: Sum of 1:50 and 51:100 in parallel

```
> library(future)
> plan(multiprocess)  # parallelize on local computer

> fa <- future( slow_sum( 1:50 ) )    # ~0 seconds
> fb <- future( slow_sum(51:100) )    # ~0 seconds
> mean(1:3)
[1] 2

> a <- value(fa)                      # blocks until ready
> b <- value(fb)
> a + b
[1] 5050
```

User chooses how to parallelize - many options

```r
plan(sequential)

plan(multiprocess)

plan(cluster, workers=c("n1", "n2", "n3"))

plan(cluster, workers=c("n1", "m2.uni.edu", "vm.cloud.org"))

plan(batchtools_slurm)     # on a Slurm job scheduler

plan(future.callr::callr)  # locally using callr

...
```

Building things using the core future blocks

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

A parallel version of lapply()

```r
#' @importFrom future future value
parallel_lapply <- function(X, FUN, ...) {
  # Create futures
  fs <- lapply(X, function(x) future(FUN(x, ...)))
  # Collect their values
  lapply(fs, value)
}

> plan(multiprocess)
> X <- list(a = 1:50, b = 51:100, c = 101:150)
> y <- parallel_lapply(X, slow_sum)              # 1 minute
> str(y)
List of 4
 $ a: int 1275
 $ b: int 3775
 $ c: int 6275
```

R package: future.apply

CRAN 1.3.0

- Futurized version of base R's `lapply()`, `vapply()`, `replicate()`, ...
- ... on all future-compatible backends
- Load balancing ("chunking")
- Proper parallel random number generation

```
y <-        lapply(X, slow_sum)
y <- future_lapply(X, slow_sum)
```

```
plan(multiprocess)
plan(cluster, workers=c("n1", "n2", "n3"))
plan(batchtools_slurm)
...
```

- Other higher-level packages: **foreach** w/ **doFuture,** and **furrr**

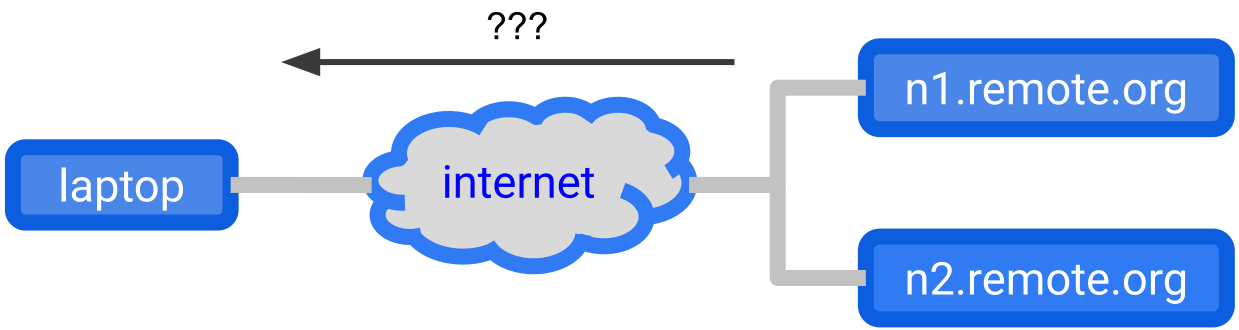
WISH: Progress bars?

ME:

Progress bars + parallel processing = complicated

How do we communicate progress from workers to main R?

- A progress bar is displayed in our main R session
- Our parallel code may be executed on external machines



How to make sure it works the same everywhere?

- Futures must work the same regardless how and if you parallelize
- We don't know how and where users will parallelize

Progress bars prevent inclusive design

- Different packages display progress different
- Progress presentation is frozen at development
- User has little control over presentation
- **Screen readers struggle with progress bars in the terminal**

```
|===========                                |   40%
```

updates

Progress ~~bars~~

Separate APIs for developers and users

API for Developers

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

Developer decides:

where in the code progress updates should be signaled

API for End Users

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

User decides:

if, when, and how progress updates are presented

Developer focuses on providing updates

Package code

```
slow_sum <- function(x) {
  p <- progressor(along=x)
  sum <- 0
  for (k in seq_along(x)) {
    Sys.sleep(0.1)
    sum <- sum + x[k]
    p(paste("Add", x[k]))
  }
  sum
}
```

User

```
> x <- 1:10
> y <- slow_sum(x)
> y
[1] 55

# progress updates
> with_progress(y <- slow_sum(x))
|=======                    |  40%
```

User choses how progress is presented

```
# without progress updates
x <- 1:10
y <- slow_sum(x)
```

```
handlers("txtprogressbar")
with_progress(y <- slow_sum(x))
|==========                        |  40%
```

```
handlers("progress")
with_progress(y <- slow_sum(x))
[======>------------]  40% Add 4
```

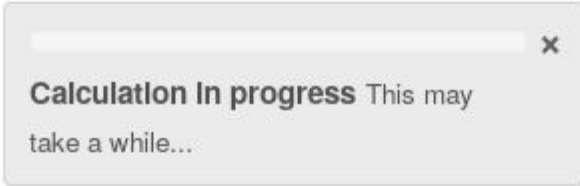
```
handlers("beepr")
with_progress(y <- slow_sum(x))
♬ ♪ ♪ ♪ … ♬
```

```
handlers("progress", "beepr")
with_progress(y <- slow_sum(x))
[======>------------]  40% Add 4
♬ ♪ ♪ ♪ … ♬
```

```
# Easy to develop new ones:
handlers("rstudio")
handlers("shiny")
handlers("pushbullet")
```

future + progressr - it just works

```r
with_progress({
  p <- progressor(along=x)
  y <- future_lapply(x, function(i) {
    p()
    ...
  })
})
```



To be decided: Should `future_lapply()` and likes auto-signal progression?

```r
with_progress({
  y <- future_lapply(x, function(i) { ... })
})
```

Exciting news: future + PROGRESS “v2” = should work

CRAN package **progress**:

```
progress::progress_bar$new(...)
```

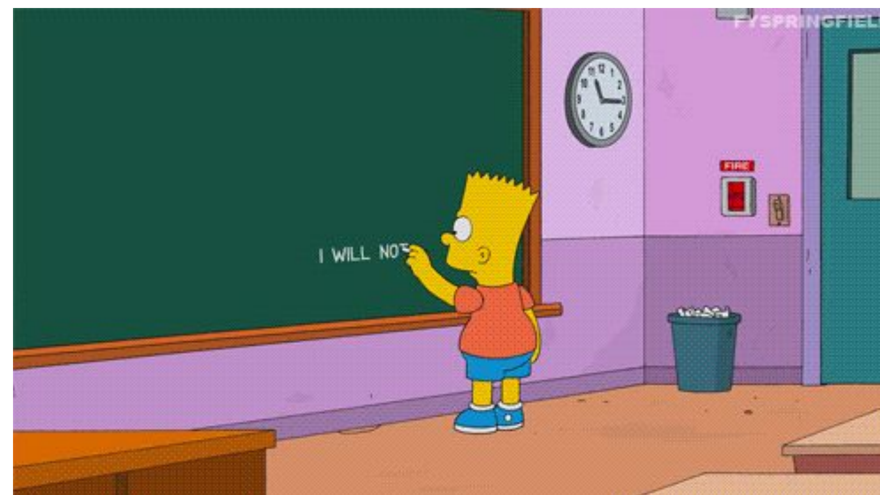
Gábor Csárdi has *work in progress* that will separate the Developer API and the End-user API; *[“PARAPHRASING”]*

- `p <- progress$new(...)`
- `p$tick()  # signal a progress condition`

This works because futures are invariant to the progress implementation!

Take home: future = worry-free parallelization

- "Write once, run anywhere" - your code is future proof
- Global variables - automatically taken care of
- Stdout, messages, warnings, *progress* - captured and relayed
- User can leverage their compute resource, e.g. compute clusters
- Atomic building blocks for higher-level parallelization APIs
- 100% cross platform code



Building a better future

I 💜 feedback, bug reports, and suggestions

@HenrikBengtsson
HenrikBengtsson/future

Thank you all!

Acknowledgments:

- Organizers, Volunteers, and Sponsors
- R Core, CRAN, devels, and users!
- R Consortium
- Gábor Csárdi