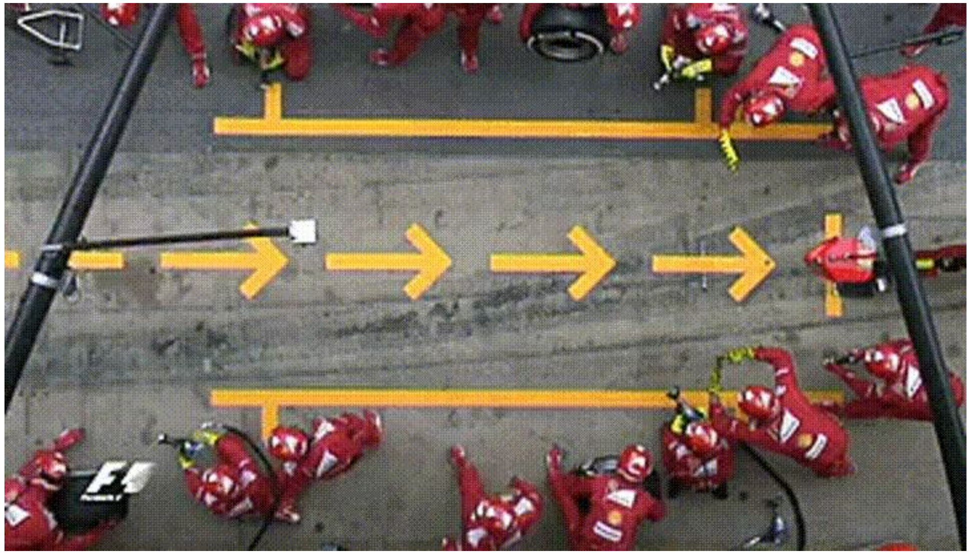
Future: Simple, Friendly Parallel Processing for R



Henrik Bengtsson
University of California San Francisco
R Foundation, R Consortium

@HenrikBengtsson
HenrikBengtsson
jottr.org

New York Open Statistical Programming Meetup on 2020-11-09

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

History - What's Already Available in R?

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)
y <- parLapply(X, slow_sum, cl=workers)     # 1 minute
```

Things we need to be aware of

mclapply() - pros and cons

Pros:

- `mclapply()` works just like `lapply()`
- `mclapply()` comes with all R installations
- no need to worry about global variables and loading packages

Cons:

- forked processing => not supported on MS Windows
- forked processing => does not work well with multi-threaded code and GUIs, e.g. may core dump RStudio

Use forked processing with care

R Core & mclapply author Simon Urbanek wrote on R-devel (April 2020):

“Do NOT use mcparallel() in packages except as a non-default option that user can set ... Multicore is intended for HPC applications that need to use many cores for computing-heavy jobs, but it does not play well with RStudio and more importantly you [as the developer] don't know the resource available so only the user can tell you when it's safe to use.”

parLapply() - pros and cons

Pros:

- `parLapply()` works just like `lapply()`
- `parLapply()` comes with all R installations
- `parLapply()` works on all operating systems

Cons:

- Requires manually loading of packages on workers
- Requires manually exporting globals to workers

Average Height of Humans and Droids

```
> library(dplyr)
> starwars[, c(1:3,10:11)]
# A tibble: 87 x 5
   name                  height  mass homeworld species
   <chr>                  <int> <dbl> <chr>     <chr>
 1 Luke Skywalker           172    77 Tatooine  Human
 2 C-3PO                    167    75 Tatooine  Droid
 3 R2-D2                     96    32 Naboo     Droid
 4 Darth Vader              202   136 Tatooine  Human
 5 Leia Organa              150    49 Alderaan  Human
 6 Owen Lars                178   120 Tatooine  Human
 7 Beru Whitesun lars       165    75 Tatooine  Human
 8 R5-D4                     97    32 Tatooine  Droid
 9 Biggs Darklighter        183    84 Tatooine  Human
10 Obi-Wan Kenobi           182    77 Stewjon   Human
# … with 77 more rows
```

parLapply() - packages must be loaded

```
library(dplyr)
y <- lapply(c("Droid", "Human"), function(kind) {
  mean(filter(starwars, species == kind)$height, na.rm=TRUE)
})
unlist(y)
## [1] 131.2000 176.6452
```

```
y <- parLapply(cl, c("Droid", "Human"), function(kind) {
  mean(filter(starwars, species == kind)$height, na.rm=TRUE)
})
## Error in checkForRemoteErrors(val) :
## 2 nodes produced errors; first error: object 'starwars' not found
```

parLapply() - packages must be loaded

```
clusterEvalQ(cl, library(dplyr))    # load 'dplyr' on all workers

y <- parLapply(cl, c("Droid", "Human"), function(kind) {
  mean(filter(starwars, species == kind)$height, na.rm=TRUE)
})
unlist(y)
## [1] 131.2000 176.6452
```

parLapply() - globals must be exported

```r
avg_height <- function(data, kind) {
  mean(filter(data, species == kind)$height, na.rm=TRUE)
}

clusterEvalQ(cl, library(dplyr))  # load 'dplyr' on all workers

y <- parLapply(cl, c("Droid", "Human"), function(kind) {
  avg_height(starwars, kind)
})

## Error in checkForRemoteErrors(val) : 2 nodes produced
## errors; first error: could not find function "avg_height"
```

parLapply() - globals must be exported

```r
avg_height <- function(data, kind) {
  mean(filter(data, species == kind)$height, na.rm=TRUE)
}

clusterEvalQ(cl, library(dplyr)) # load 'dplyr' on all workers
clusterExport(cl, "avg_height")  # export function to all workers
y <- parLapply(cl, c("Droid", "Human"), function(kind) {
  avg_height(starwars, kind)
})
unlist(y)
## [1] 131.2000 176.6452
```

Design patterns found in CRAN packages

My customize sum function

```
total_slow_sum <- function(X) {
  y <- lapply(X, slow_sum)
  sum(unlist(y))
}

> X <- list(a=1:50, b=51:100, c=101:150)
> y <- total_slow_sum(X)
> y
[1] 11325
```

v1. A first attempt on parallel support

```r
#' @importFrom parallel mclapply detectCores
total_slow_sum <- function(X, parallel = FALSE) {
  if (parallel) {
    y <- mclapply(X, slow_sum, mc.cores = detectCores())
  } else {
    y <- lapply(X, slow_sum)
  }
  sum(unlist(y))
}
```

```
> y <- total_slow_sum(X, parallel = TRUE)
> y
[1] 11325
```

v2. A slightly better approach

```r
total_slow_sum <- function(X, parallel = FALSE) {
  if (parallel) {
    y <- mclapply(X, slow_sum) # Better; user decides number of cores
  } else {
    y <- lapply(X, slow_sum)
  }
  sum(unlist(y))
}
```

```
> options(mc.cores = 4)
> y <- total_slow_sum(X, parallel = TRUE)
> y
[1] 11325
```

v3. An alternative approach

```
total_slow_sum <- function(X, ncores = 1) {
  if (ncores > 1) {
    y <- mclapply(X, slow_sum, mc.cores = ncores)
  } else {
    y <- lapply(X, slow_sum)
  }
  sum(unlist(y))
}

> y <- total_slow_sum(X, ncores = 4)
> y
[1] 11325
```

v4. Support also MS Windows

```r
total_slow_sum <- function(X, ncores = 1) {
  if (ncores > 1) {
    if (.Platform$OS.type == "windows") {
      cl <- makeCluster(ncores)
      on.exit(stopCluster(cl))
      clusterEvalQ(cl, library(somepkg))
      clusterExport(cl, "some_global")
      y <- parLapply(X, slow_sum)
    } else {
      y <- mclapply(X, slow_sum, mc.cores = ncores)
    }
  } else {
    y <- lapply(X, slow_sum)
  }
  sum(unlist(y))
}
```

- *Can you please add support for AAA parallelization too?*
- *While you’re at it, what about BBB parallelization?*

v99: Phew ... will this do?

```r
total_slow_sum <- function(X, parallel = "none") {
  if (parallel == "snow") {
    cl <- getDefaultCluster()
    clusterEvalQ(cl, library(somepkg))
    clusterExport(cl, "some_global")
    y <- parLapply(cl, X, slow_sum)
  } else if (parallel == "multicore") {
    y <- mclapply(X, slow_sum)
  } else if (parallel == "clustermq") {
    y <- clustermq::Q(slow_sum, X,
           pkgs="somepkg", export="some_global")
  } else if (parallel == ...) {
    ...
  } else {
    y <- lapply(X, slow_sum)
  }
  sum(unlist(y))
}
```

What's my test coverage now?

- *There is this new, cool DDD parallelization method … ?*
- *…*
- *Still there?*

PROBLEM: Different APIs for different parallelization strategies

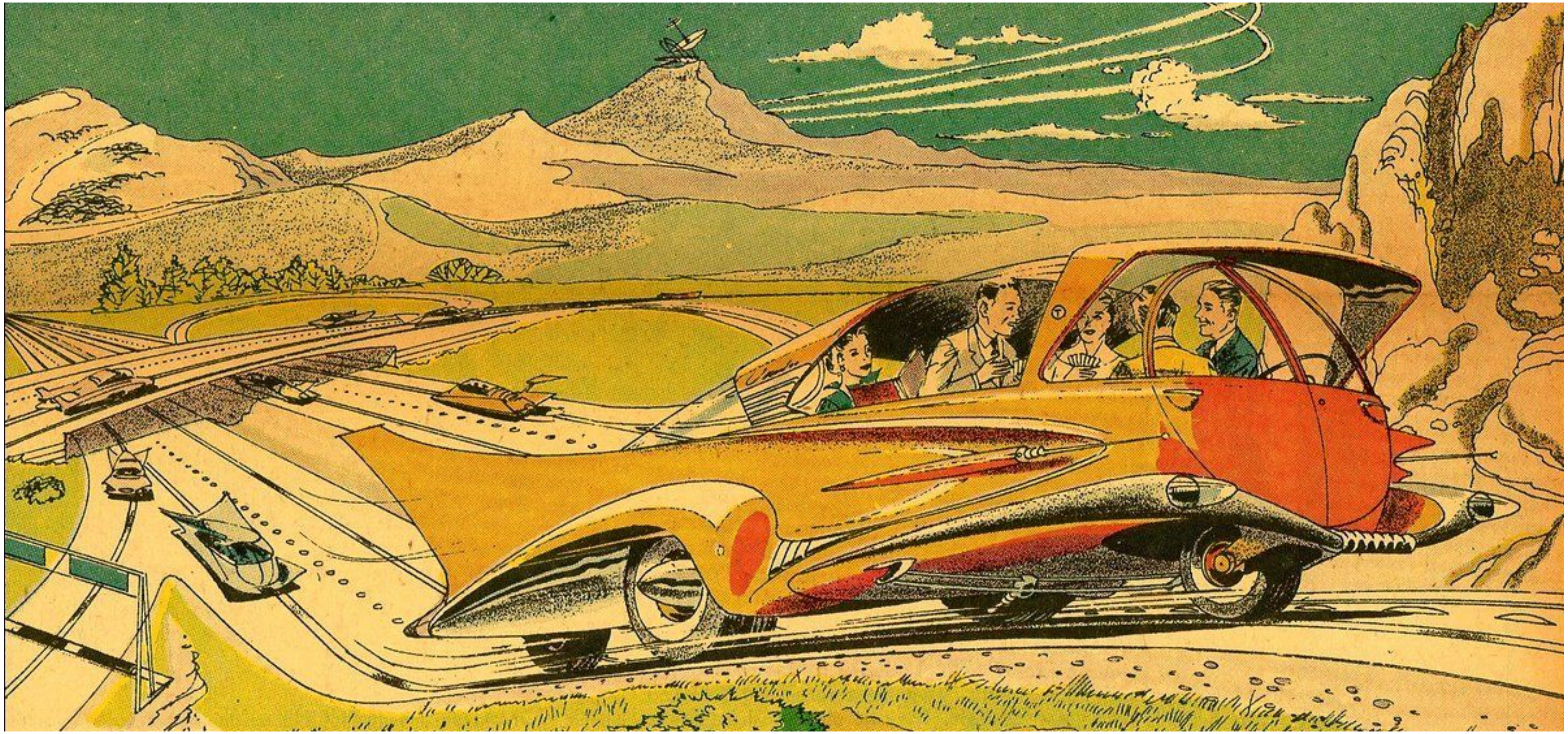
Developer:

- 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 run in parallel
- 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

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

Dan LaBar
@embiggenData

“Low friction”:

- automatically exports **global variables**
- automatically relays output, messages, and warnings
- proper parallel random number generation (RNG)

A Future is ...

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

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(multisession)  # 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                               # here at ~1 minute
[1] 5050
```

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

```
> library(future)
> plan(multisession)

> a %<-% slow_sum( 1:50 )
> b %<-% slow_sum(51:100)
> mean(1:3)
[1] 2

> a + b                                   # blocks until ready
[1] 5050
```

User chooses how to parallelize - many options

```
plan(sequential)

plan(multicore)                # uses the mclapply() machinery

plan(multisession)             # uses the parLapply() machinery

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

...
```

Globals automatically identified (99% worry free)

Static-code inspection by walking the abstract syntax tree (AST):

```
x <- rnorm(n = 100)               pryr::ast( { slow_sum(x) } )
f <- future({ slow_sum(x) })         |  \- `{
            \_____________/          |  \- ()
                   |_____________|    \- `slow_sum
                                     |    \- `x
```

=> globals & packages identified and exported to the worker:

- `slow_sum()` - a function (also searched recursively)
- `x` - a numeric vector of length 100

Comment: Globals & packages can also be specified manually

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(multisession)
> 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

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

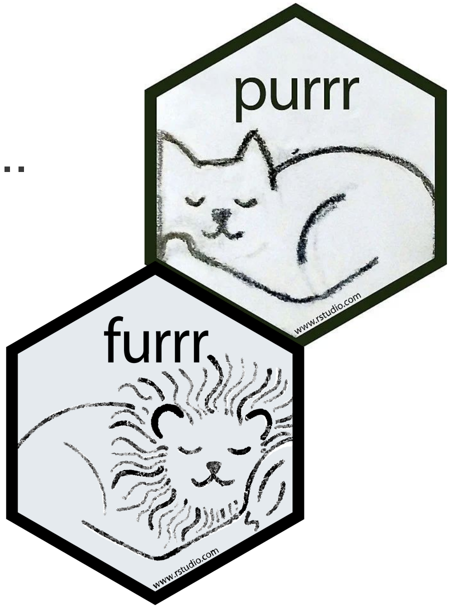
```
y <-        lapply(X, slow_sum)
y <- future_lapply(X, slow_sum)
```

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

R package: furrr (Davis Vaughan)

- Futurized version of **purrr**'s `map()`, `map2()`, `modify()`, ...
- ... on all future-compatible backends
- Load balancing ("chunking")
- Proper parallel random number generation

```
y <-        map(X, slow_sum)
y <- future_map(X, slow_sum)
```



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

R package: doFuture

- Futurized foreach adaptor
- ... on all future-compatible backends
- Load balancing ("chunking")

```
y <- foreach(x = X) %do% slow_sum(x)
doFuture::registerDoFuture()
y <- foreach(x = X) %dopar% slow_sum(x)



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

Stay with your favorite coding style

```r
# Base R style (R & future.apply)
y <- lapply(X, slow_sum)
y <- future_lapply(X, slow_sum)

# Tidyverse style (purrr & furrr)
y <- X %>% map(slow_sum)
y <- X %>% future_map(slow_sum)

# Foreach style (foreach & doFuture)
y <- foreach(x = X) %do% slow_sum(x)
y <- foreach(x = X) %dopar% slow_sum(x)
```

Output, Warnings, and Errors

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
>
```

- Output and conditions are displayed just like `lapply()`
- This does not work when using `mclapply()` or `parLapply()`

Standard output is truly relayed

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

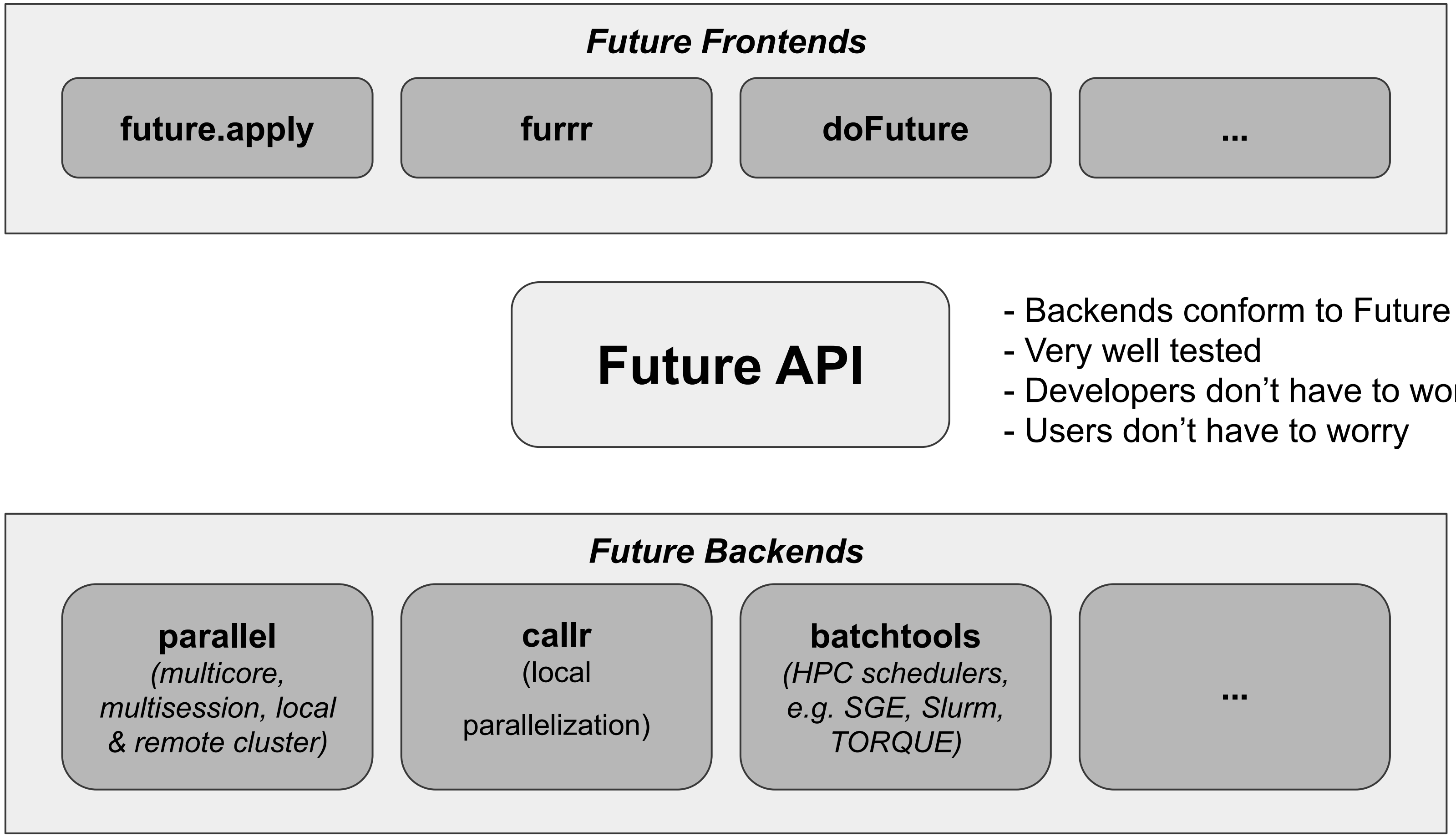
Conditions are truly relayed

```
x <- c(-1, 10, 30)
tryCatch({
  y <- future_lapply(x, function(z) {
    log(z)
  })
}, warning = function(w) {
  # Bump warning to an error
  stop(conditionMessage(w), call.=FALSE)
})

Error: NaNs produced
```

Can I trust the future framework?

Future API guarantees uniform behavior



- Backends conform to Future API
- Very well tested
- Developers don’t have to worry
- Users don’t have to worry

Large amounts of testing ... all the time

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

Tested on Linux, macOS, Solaris, Windows
Tested on old and new versions of R
Revdep checks on > 140 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)

```
$ Rscript
```

Not everything can be parallelized

Some objects cannot be exported to another R process

```r
plan(multisession)
file <- tempfile()
con <- file(file, open="wb")
cat("hello", file = con)
f <- future({ cat("world", file = con); 42 })
v <- value(f)
## Error in cat("world", file = con) : invalid connection
```

Note, this is true for all parallelization frameworks. There’s no solution to this.

Non-exportable objects

For troubleshooting, ask the future framework to look for non-exportable objects:

```
options(future.globals.onReference = "error")
file <- tempfile()
con <- file(file, open="wb")
cat("hello", file = con)
f <- future({ cat("world", file = con); 42 })
## Error: Detected a non-exportable reference ('externalptr')
## in one of the globals ('con' of class 'file') used in the
## future expression
```

Disabled by default because (i) some false positive, but also (ii) expensive.

Less obvious, non-exportable objects

```
library(xml2)
xml <- read_xml("<body></body>")
f <- future({ xml_children(xml) })
value(f)
## Error: external pointer is not valid

str(xml)
## List of 2
##  $ node:<externalptr>
##  $ doc :<externalptr>
##  - attr(*, "class")= chr [1:2] "xml_document" "xml_node"
```

Roadmap - what on the horizon?

Terminating futures, if backend supports it

If supported by the parallel backend, free up worker by terminating futures no longer of interest, e.g.

```r
plan(multisession)
f1 <- future({ very_slow(x) })
f2 <- future({ also_slow(x) })
if (resolved(f2)) {
  y <- value(f2)
  # First future no longer needed; attempt to terminate it
  discard(f1)
}
```

Bonus: Automatically register a finalize so that removed futures call **`discard()`** on themselves when garbage collector.

Terminate, exit early from map-reduce calls

With terminate, we can also terminate useless futures in parallel map-reduce, and then exit early, e.g.

```
plan(multisession)
X <- list(42, 31, "pi”, pi)
y <- future_lapply(X, slow)
## Error in ...future.FUN(...future.X_jj, ...) :
##   non-numeric argument to mathematical function
```

Today:
The error is not thrown until all slow(41), slow(31), and slow(pi) finish

Idea:
As soon as the error is detected, terminate all running futures, and rethrow error

Exception handling on extreme events

If there is an extreme event such as a power outage of a machine where one of the futures are currently resolved, the future framework detects this;

```
> f <- future(quit("no"))
> value(f)
Error in unserialize(node$con) :
  Failed to retrieve the value of MultisessionFuture (<none>) from
cluster RichSOCKnode #1 (PID 25562 on localhost 'localhost'). The
reason reported was 'error reading from connection'. Post-mortem
diagnostic: No process exists with this PID, i.e. the localhost
worker is no longer alive.
```

Exception handling on extreme events

We can handle these exception at the very lowest level:

```
tryCatch({
  v <- value(f)
}, FutureError = function(ex) {
  # Do something, e.g. restart worker and relaunch future
})
```

Open questions:

- How to relaunch a future?
- How to restart a worker - whatever that means?
- How should this work for map-reduce APIs, e.g. **future.apply** and **furrr**?

Prepopulate workers with data

```
library(parallel)
cl <- makeCluster(4)
huge <- very_large_object()

clusterExport(cl, "huge")          # Export only once

fit1 <- parLapply(params_1, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
})

fit2 <- parLapply(params_2, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
})
```

By design, the Future API does not have a concept of a specific worker

There is no method for exporting an object to all workers. Thus, ‘huge’ is exported to the workers twice:

```
fit1 <- future_lapply(params_1, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
})

fit2 <- future_lapply(params_2, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
})
```

Sticky globals - avoid repeated exports

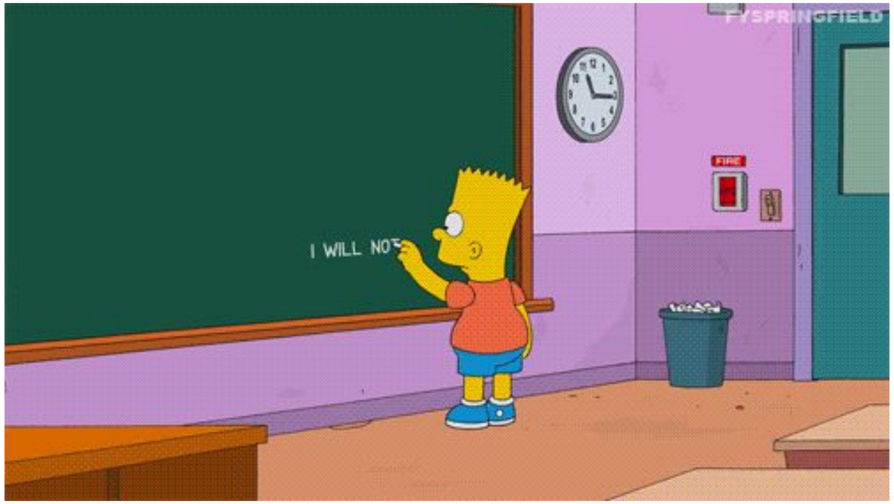
```
fit1 <- future_lapply(params_1, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
}, future.cache = "huge")

fit2 <- future_lapply(params_2, function(offset) {
  lm(y ~ x + 1, data = huge - offset))
})
```

- Might be ignored: Not all backends support caching
- Backend API: Identify which workers have verbatim object cached
- Workers might have garbage collected their cache
- What should happen if only cached on a busy worker?

Take home: future = 99% 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

Thank you all!

@HenrikBengtsson
HenrikBengtsson
jottr.org

Q & A

Q. Nested parallelization?

E.g. one individual per machine then one chromosome per core:

```
nodes <- c(rep("n1", 4), rep("n2", 2), "remote.org", "server.cloud")
plan(list(tweak(cluster, workers = nodes), multsession))

fastq <- dir(pattern = "[.]fq$")
bam <- listenv()
for (i in seq_along(fastq)) {
  ## One individual per worker
  bam[[i]] %<-% {
    chrs <- listenv()
    for (j in 1:24) {
      ## One chromosome per core
      chrs[[j]] %<-% DNAseq::align(fastq[i], chr = j)
    }
    merge_chromosomes(chrs)
  }
}
```

Q. Why not detectCores()?

Don't hard-code the number of workers to parallelize, e.g.

```
myfcn <- function(X) {
  y <- mclapply(X, slow, mc.cores = parallel::detectCores())
}
```

This is a bad idea because:

- as a developer we do not know where the user will run this
- user might run to R processes calling `myfcn()` at the same time
- there might be other users on the same machine
- `myfcn()` might be called by another function already running in parallel

availableCores() instead of detectCores()

`parallel::detectCores()`

- may return `NA_integer_`
- uses all cores; ignores other settings
- two or more users doing this on the same machine, will overwhelm the CPUs

R Core & mclapply author Simon Urbanek wrote on R-devel (April 2020):
*"**Multi-core machines are often shared so using all detected cores is a very bad idea**. The user should be able to explicitly enable it, but it should not be enabled by default."*

`future::availableCores()`

- always returns >= 1
- defaults to `parallel::detectCores()` but respects also other settings, e.g. `MC_CORES`, HPC scheduler environment variables, …
- sysadms can set the default to a small number of cores via an env variable

Parallelize using all but one core?

A very common meme is to use all but cores

```
ncores <- detectCores()-1
```

The idea is that we save one CPU core so we can keep working on the computer.

However, note that your code might run a single-core machine, which result in ncores = 0. To account for this and the missing value, use:

```
ncores <- max(1, detectCores()-1, na.rm=TRUE)
```

But, again, it’s better to use:

```
ncores <- max(1, availableCores()-1)
```

Q. How to detect non-exportable globals?

Some objects only works in the R session where they were created. If exported to another R process, they will either give an error when used, or garbage results.

There is no clear method for identify objects that fail when exported. However, objects with “external pointer” (`<externalptr>`) used by native code often fail, although not all of them (e.g. data.table object).

To detect external pointer and other types of “references”, the future package (optionally) scans the globals using something like:

```
con <- file(nullfile(), open = "wb")
serialize(globals, connection = con, refhook = function(ref) {
  stop("Detected a reference")
})
```

Q. Should I use doParallel or doFuture?

The foreach adaptor **doParallel** supports two types of parallel backends. Which one we get how we call `registerDoParallel()` and from what operating system.

1. "cluster": uses the `parLapply()` machinery

   ```r
   # All operating systems
   cl <- makeCluster(4)
   registerDoParallel(cl)

   # On MS Windows one can also do
   registerDoParallel(4)
   ```

2. "multicore": uses the `mclapply()` machinery

   ```r
   # On Linux, Solaris, macOS (only)
   registerDoParallel(4)
   ```

doFuture can parallelize like doParallel & more

The foreach adaptor **doFuture** supports *any* type of parallel backend that the future framework support, including the two “cluster” and “multicore” that **doParallel** supports. Here’s how they’re related:

	doFuture	**doParallel**

cluster	`# All operating systems`
	`registerDoFuture()`
	`cl <- makeCluster(4)`
	`plan(cluster, workers=cl)`
	
	`# All operating system`
	`registerDoFuture()`
	`plan(multisession, workers=4)`
		`# All operating systems`
		`cl <- makeCluster(4)`
		`registerDoParallel(cl)`
		
		`# MS Windows`
		`registerDoParallel(4)`

multicore	`# On Linux, macOS, Solaris (only)`
	`registerDoFuture()`
	`plan(multicore, workers=4)`
		`# On Linux, macOS, Solaris (only)`
		`registerDoParallel(4)`

Extra features that comes with doFuture

There is *no* performance difference between **doFuture** and **doParallel** when using “multicore” or “multisession” backends; they both rely on the same parallelization frameworks in the **parallel** package.

The main advantage of using the **doFuture** over **doParallel** is that standard output, messages, and warnings are relayed to the main R session.

doFuture	**doParallel**
`registerDoFuture()`	`cl <- makeCluster(2)`
`cl <- makeCluster(2)`	`registerDoParallel(cl)`
`plan(cluster, workers = cl)`	
`y <- foreach(x = 1:2) %dopar% {`	`y <- foreach(x = 1:2) %dopar% {`
`  message("x = ", x)`	`  message("x = ", x)`
`  sqrt(x)`	`  sqrt(x)`
`}`	`}`
`## x = 1`	
`## x = 2`	*Output is not relayed!*