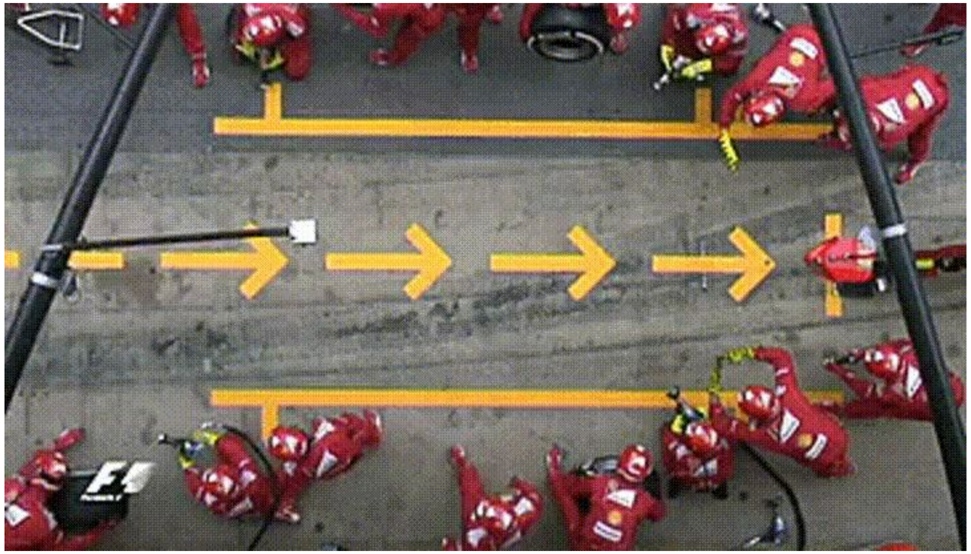
Future: A Simple, Extendable, Generic Framework for Parallel Processing in R



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

@HenrikBengtsson
HenrikBengtsson
jottr.org

Edmonton R User Group Meetup on 2023-05-22

We parallelize software for various reasons

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 UI

We parallelize software for various reasons

We may choose to parallelize on:

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

History - What’s Already Available in R?

R comes with built-in parallelization

```
library(DNAseq)
fq <- c("a.fq", "b.fq", "c.fq")                  # In: FASTQ files
bam <- lapply(fq, align)                         # 3 hours
## [1] "a.bam" "b.bam" "c.bam"                   # Out: BAM files
```

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

```
library(parallel)
bam <- mclapply(fq, align, mc.cores = 3)         # 1 hour
```

To parallelize also on Windows, we can do:

```
library(parallel)
workers <- makeCluster(3)
clusterEvalQ(workers, library(DNAseq))
bam <- parLapply(fq, align, cl = workers)        # 1 hour
```

Things we need to be aware of

mclapply() - magic with problems

Pros:

- `mclapply()` works *similarly* to `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 => unstable with *multi-threaded* code & GUIs, e.g. may core dump RStudio
- errors have to be handled with great care

⚠️ Use forked processing with care!

R Core & `mclapply()` author Simon Urbanek ([on R-devel, 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() - takes some efforts

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, e.g.
  `clusterEvalQ(workers, library(DNAseq))`
- Requires manually exporting globals to workers, e.g.
  `clusterExport(workers, c("varA", "varB"))`
- errors have to be handled with care

Design patterns found in packages

My “align them all” function

```
align_all <- function(fq) {
  lapply(fq, align)
}
```

```
> fq <- c("a.fq", "b.fq", "c.fq")
> bam <- align_all(fq)
> bam
[1] "a.bam" "b.bam" "c.bam"
```

v1. A first attempt on parallel support

```r
align_all <- function(fq, parallel = FALSE) {
  if (parallel) {
    bam <- mclapply(fq, align, mc.cores = availableCores())
  } else {
    bam <- lapply(fq, align)
  }
  bam
}

> bam <- align_all(fq, parallel = TRUE)
> bam
[1] "a.bam" "b.bam" "c.bam"
```

v2. A slightly better approach

```r
align_all <- function(fq, parallel = FALSE) {
  if (parallel) {
    bam <- mclapply(fq, align) # let user decide on cores!👍
  } else {
    bam <- lapply(fq, align)
  }
  bam
}


> options(mc.cores = 4)
> bam <- align_all(fq, parallel = TRUE)
```

v3. Yet another alternative

```
align_all <- function(fq, ncores = 1) {
  if (ncores > 1) {
    bam <- mclapply(fq, align, mc.cores = ncores)
  } else {
    bam <- lapply(fq, align)
  }
  bam
}

> bam <- align_all(fq, ncores = 4)
```

v4. Support also MS Windows

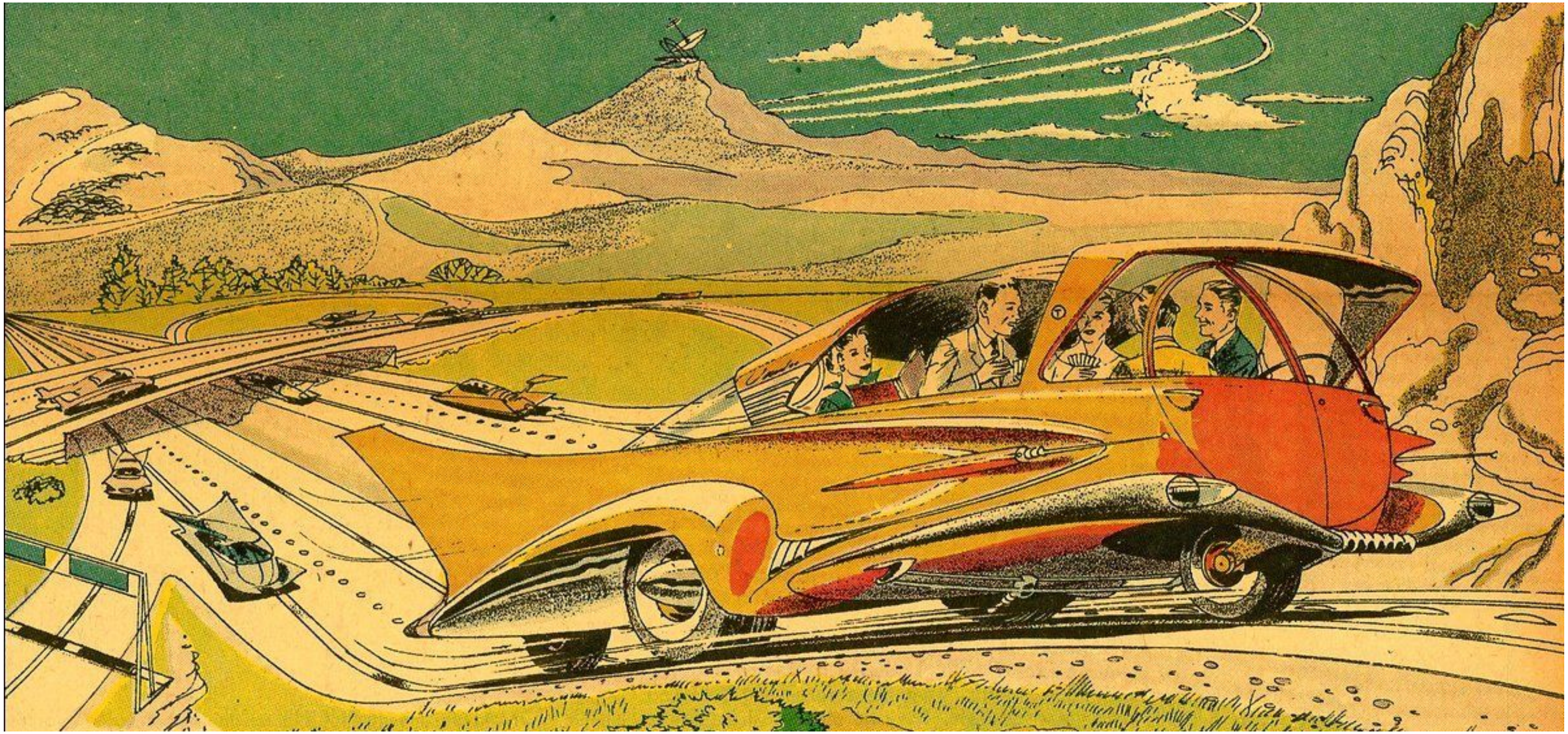
```
align_all <- function(fq, ncores = 1) {
  if (ncores > 1) {
    if (.Platform$OS.type == "windows") {
      workers <- makeCluster(ncores)
      on.exit(stopCluster(workers))
      clusterEvalQ(workers, library(DNAseq))
      clusterExport(workers, "some_global")
      bam <- parLapply(fq, align, cl = workers)
    } else {
      bam <- mclapply(fq, align, mc.cores = ncores)
    }
  } else {
    bam <- lapply(fq, align)
  }
  bam
}
```

v99: Phew ... will this do?

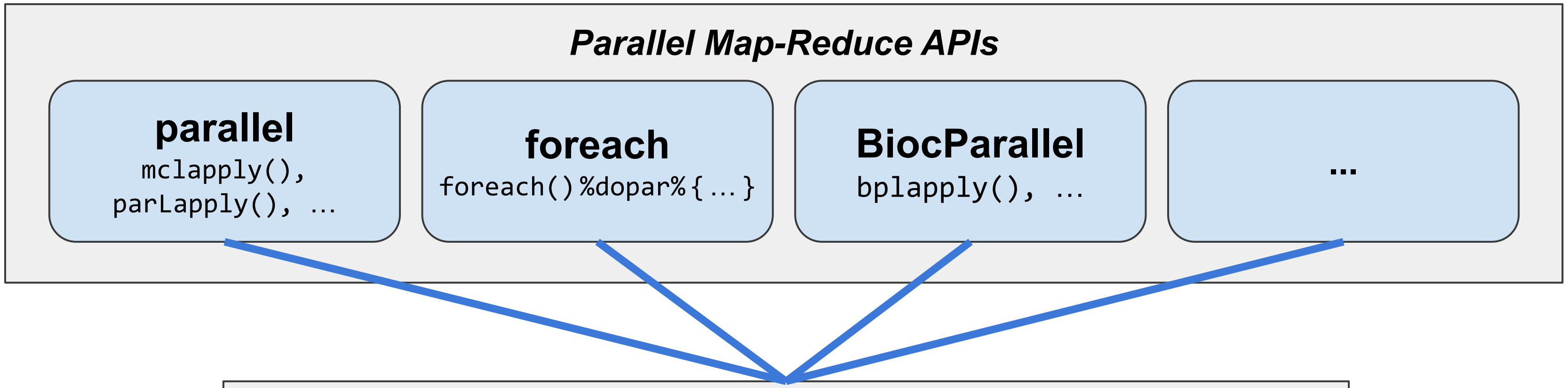
```
align_all <- function(fq, parallel = "none") {
  if (parallel == "snow") {
    workers <- getDefaultCluster()
    clusterEvalQ(workers, library(DNAseq))
    clusterExport(workers, "some_global")
    bam <- parLapply(fq, align, cl = workers)
  } else if (parallel == "multicore") {
    bam <- mclapply(fq, align)
  } else if (parallel == "clustermq") {
    bam <- clustermq::Q(align, fq,
           pkgs="DNAseq", export="some_global")
  } else if (parallel == ...) {
    ...
  } else {
    bam <- lapply(fq, align)
  }
  bam
}
```

What's my test coverage now?

Welcome to the Future



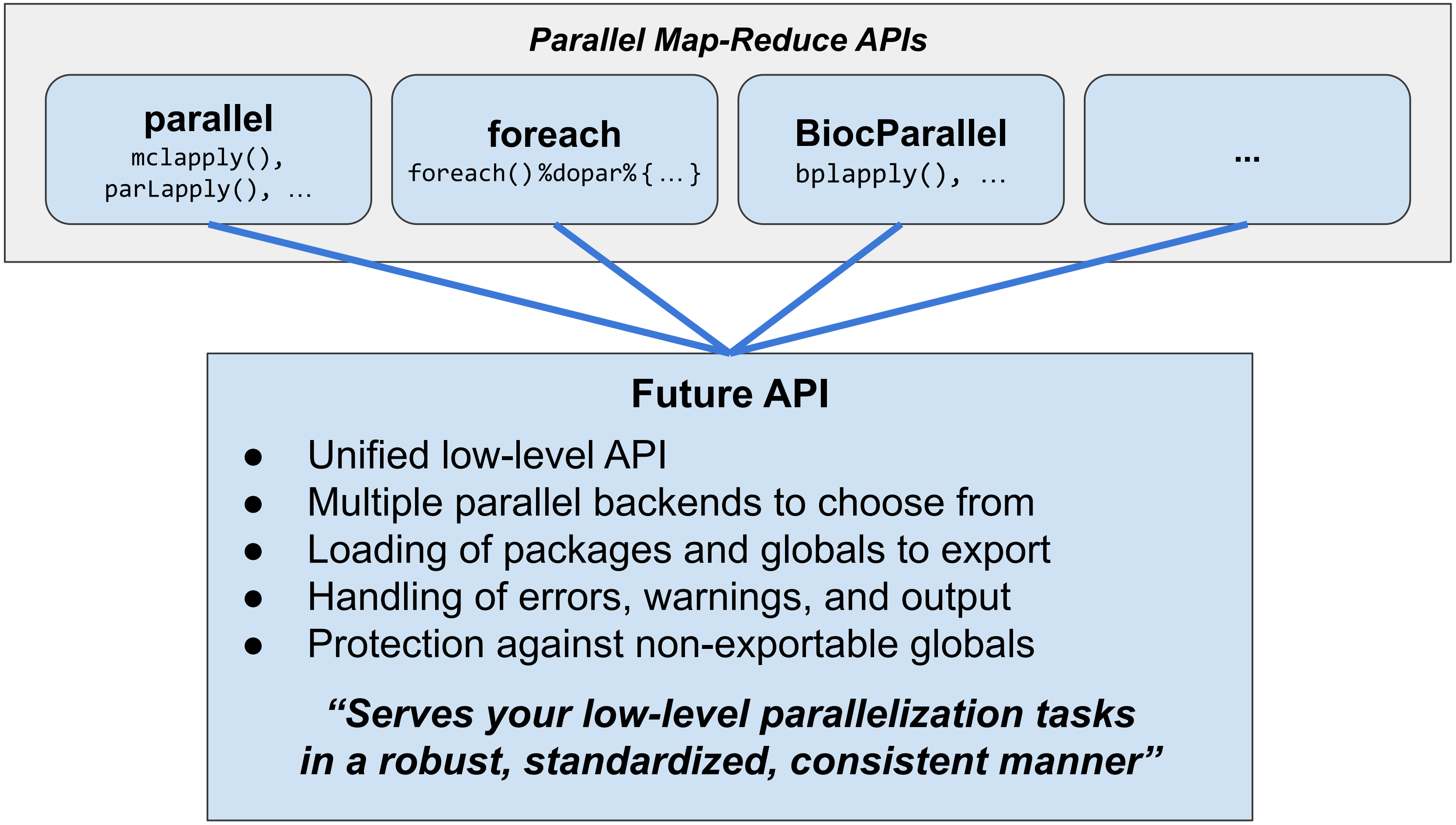
Parallel frameworks reimplement common ideas



Common needs, strategies & re-implementations:

- Familiar map-reduce functions in a unified API
- Multiple parallel backends to choose from
- Efficient iteration & chunking
- Loading of packages and globals to export
- Handling of errors, warnings, and output

Idea: Collect common tasks in one place



- Unified low-level API
- Multiple parallel backends to choose from
- Loading of packages and globals to export
- Handling of errors, warnings, and output
- Protection against non-exportable globals

"Serves your low-level parallelization tasks in a robust, standardized, consistent manner"

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

“Low friction”:

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

Dan LaBar
@embiggenData

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
```

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 package

...
```

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;

```
f <- future({ slow_sum(x) }, globals = c("slow_sum", "x"))
```

Other frameworks need manual exports

With other parallel frameworks, you have to manually export the globals that need to be available on the parallel workers, e.g.

```
library(parallel)
cl <- makeCluster(1)
x <- rnorm(n = 100)
clusterExport(cl, c("slow_sum", "x"))
y <- clusterEvalQ(cl, { slow_sum(x) })
```

Conclusion: This is *not* needed when using Futureverse for parallelization (except for rare, corner cases)

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
  value(fs)
}
```

```
> library(DNAseq)
> plan(multisession)
> bam <- parallel_lapply(fq, align)
> bam
[1] "a.bam" "b.bam" "c.bam"
```

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

```
bam <-        lapply(fq, align)
bam <- future_lapply(fq, align)
```

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

A parallel version of purrr::map()

```r
#’ @importFrom purrr map
#' @importFrom future future value
parallel_map <- function(.x, .f, ...) {
  # Create futures
  fs <- map(.x, function(x) future(.f(x, ...))
  # Collect their values
  value(fs)
}
```

```
> library(DNAseq)
> plan(multisession)
> bam <- parallel_map(fq, align)
> bam
[1] "a.bam" "b.bam" "c.bam"
```

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

```
bam <-        map(fq, align)
bam <- future_map(fq, align)
```

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

Package: doFuture

- %dofuture% - a futurized foreach adaptor
- ... on all future-compatible backends
- Load balancing ("chunking")
- Proper parallel random number generation

```
bam <- foreach(x = fq) %do%       align(x)
bam <- foreach(x = fq) %dofuture% align(x)
```

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

Stay with your favorite coding style 1/3

```
# Base R style (R & future.apply)
bam <- lapply(fq, align)
bam <- future_lapply(fq, align)

# Tidyverse style (purrr & furrr)
bam <- fq |> map(align)
bam <- fq |> future_map(align)

# Foreach style (foreach & doFuture)
bam <- foreach(x = fq) %do% align(x)
bam <- foreach(x = fq) %dofuture% align(x)
```

Stay with your favorite coding style 2/3

```
# Foreach style (classical)
doFuture::registerDoFuture()  # %dopar% to use futures
bam <- foreach(x = fq) %dopar% align(x)



# Bioconductor's BiocParallel
register(DoparParam())    # BiocParallel to use %dopar%
doFuture::registerDoFuture()  # %dopar% to use futures
bam <- bplapply(fq, align)
```

Stay with your favorite coding style 3/3

```r
# pbapply (since Jan 2023)
bam <- pblapply(fq, align, cl = "future")
```

Recall: User chooses how to parallelize

```
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 package

...
```

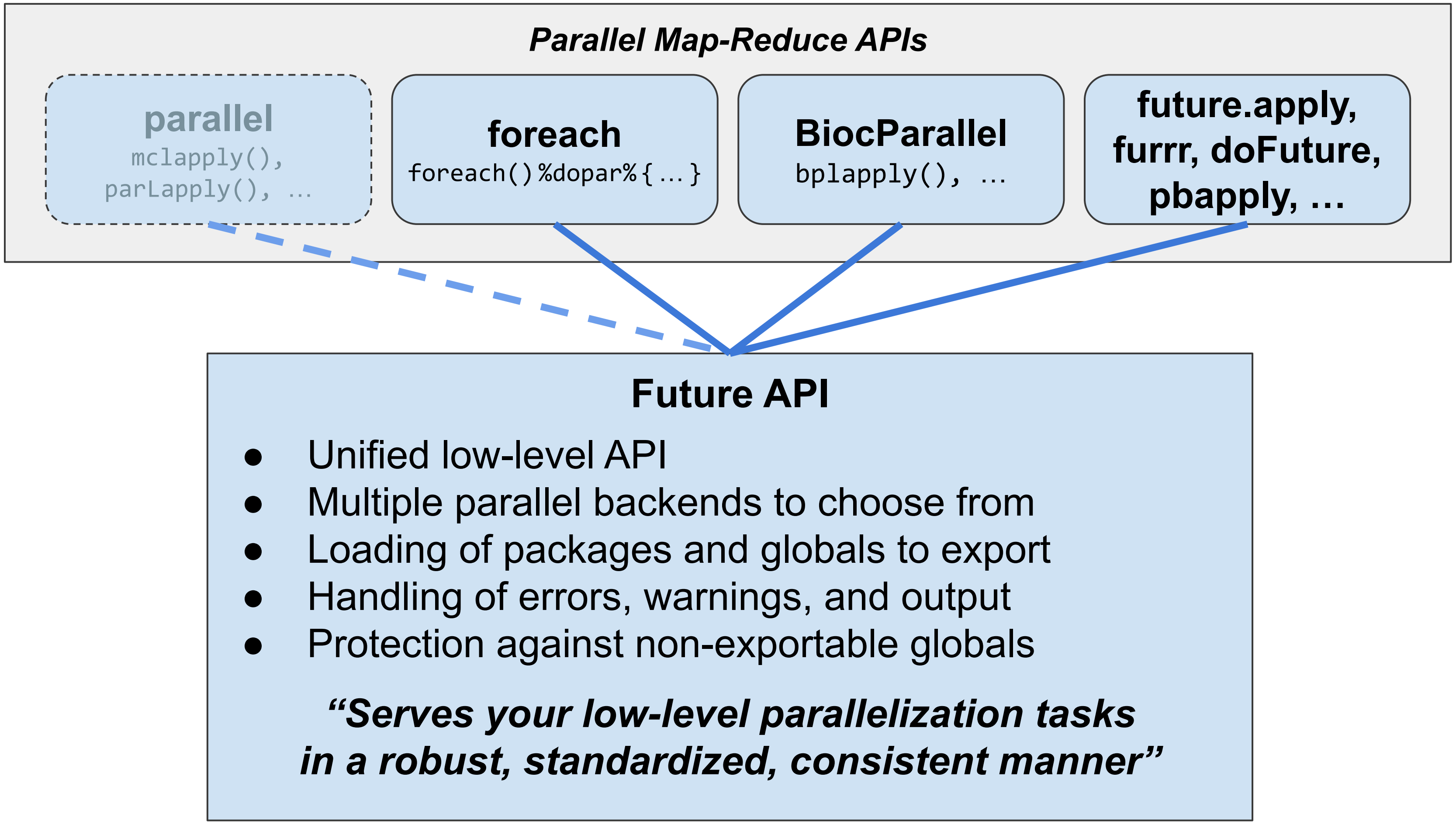
Backend package: future.batchtools

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

```
fq <- dir(pattern = "[.]fq$")          ## 80 files; 200 GB each
bam <- future_lapply(fq, align)        ## 1 hour each
```

```
{henrik: ~}$ squeue
Job ID   Name               User         Time Use S
-------- ------------------ ------------ -------- -
606411   xray               alice        46:22:22 R
606638   future_lapply-5    henrik       00:52:05 R
606641   python             bob          37:18:30 R
606643   future_lapply-6    henrik       00:51:55 R
...
```

2023: Futureverse widely supported



Parallel Map-Reduce APIs
parallel
mclapply(), parLapply(), …
foreach
foreach() %dopar% { … }
BiocParallel
bplapply(), …
future.apply, furrr, doFuture, pbapply, …
Future API

- Unified low-level API
- Multiple parallel backends to choose from
- Loading of packages and globals to export
- Handling of errors, warnings, and output
- Protection against non-exportable globals

"Serves your low-level parallelization tasks in a robust, standardized, consistent manner"

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 log(z) : NaNs produced
>
```

<= Output relayed from workers

<= Warnings are relayed too

⚠️ Other frameworks: No output/warnings

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

<= Output and warnings completely muffled!

```
> cl <- makeCluster(2)
> y <- parLapply(cl, x, function(z) {
    message("z = ", z)
    log(z)
  })
>
```

<= Output and warnings completely muffled!

⚠️ Same for foreach w/ doParallel etc.

```
> x <- c(-1, 10, 30)
> cl <- makeCluster(2)
> doParallel::registerDoParallel(cl)
> y <- foreach(z = x) %dopar% {
    message("z = ", z)
    log(z)
  }
>
```

<= Output and warnings completely muffled!

foreach w/ doFuture works

```
> x <- c(-1, 10, 30)
> y <- foreach(z = x) %dofuture% {
    message("z = ", z)
    log(z)
  }
z = -1
z = 10
z = 30
Warning message:
In log(z) : NaNs produced
>
```

<= Output relayed from workers

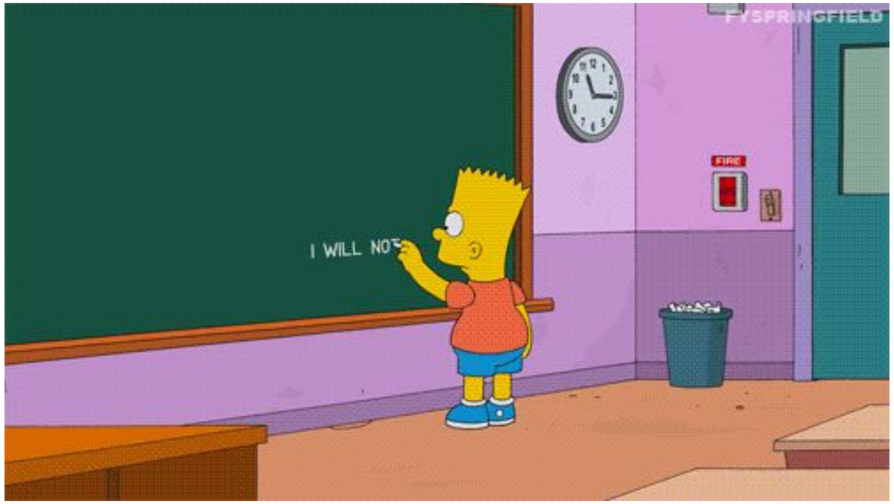
<= Warnings are relayed too

pbapply: supports futures since Jan 2023

```
> library(pbapply)
> plan(multisession)
> x <- c(-1, 10, 30)
> y <- pblapply(x, function(z) {
    message("z = ", z)
    log(z)
  }, cl = "future")
z = -1
z = 10
z = 30
  |+++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=02s
Warning message:
In log(z) : NaNs produced
>
```

Take home: future = 99% worry-free parallelization

- "Write once, run anywhere"
- 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
- Future proof: will work with still-to-be-developed backends



It’s easy to get started ❤️

- It’s easy to get started - just try it
- Support: https://github.com/HenrikBengtsson/future/discussions
- Tutorials: https://www.futureverse.org/tutorials.html
- Blog posts: https://www.futureverse.org/blog.html
- More features on the roadmap
- I love feedback and ideas

@HenrikBengtsson
HenrikBengtsson
jottr.org