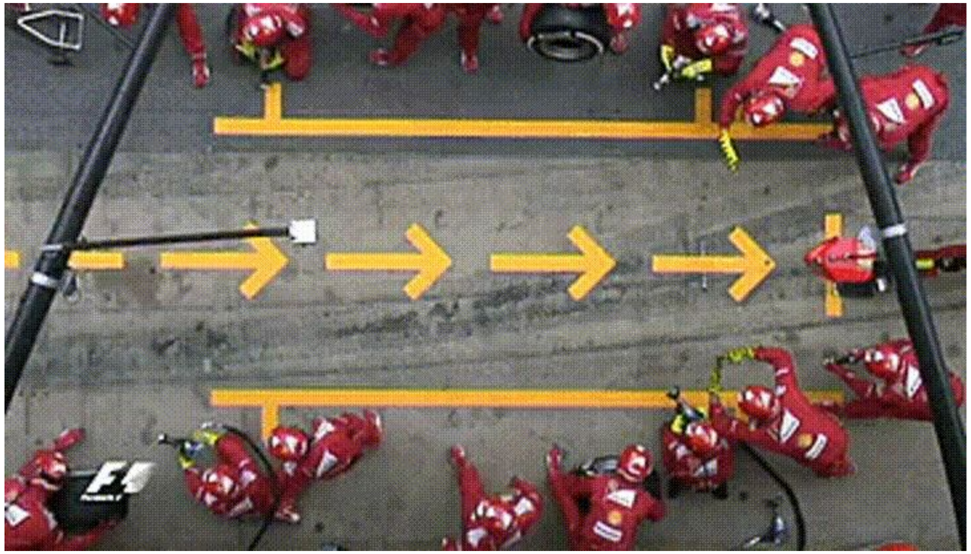
Futureverse: A Unifying Parallelization Framework in R for Everyone - Part 2



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

@HenrikBengtsson
HenrikBengtsson
jottr.org

RaukR 2024, Visby, Sweden, June 12 & 17, 2024 (https://www.futureverse.org)

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 it 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

- 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”

Dan LaBar
@embiggenData

All we need are three building blocks

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

This was invented in 1975

```
parallel_lapply <- function(X, FUN, ...) {
  fs <- lapply(X, function(x) future(FUN(x, ...))
  lapply(fs, value)
}
```

Lab 2: Refresher and parallelize purrr

- Task 1: `purrr::map() -> parallel_map()`
- Task 2: `purrr::map_dbl() -> parallel_map_dbl()`
- Task 3-4: Things that are problematic

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()

```
#' @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)
}
```

```
> library(DNAseq)
> plan(multisession)
> bam <- parallel_lapply(fq, align)
> bam
[1] "file1.bam" "file2.bam" "file3.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
  map(fs, value)
}
```

```r
> library(DNAseq)
> plan(multisession)
> bam <- parallel_map(fq, align)
> bam
[1] "file1.bam" "file2.bam" "file3.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)
...
```

“Base R vs Tidyverse”

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

# Tidyverse style (purrr & furrr)
bam <- map(fq, align)
bam <- future_map(fq, align)
```

Seriously …

It’s not a war - use the style you prefer!

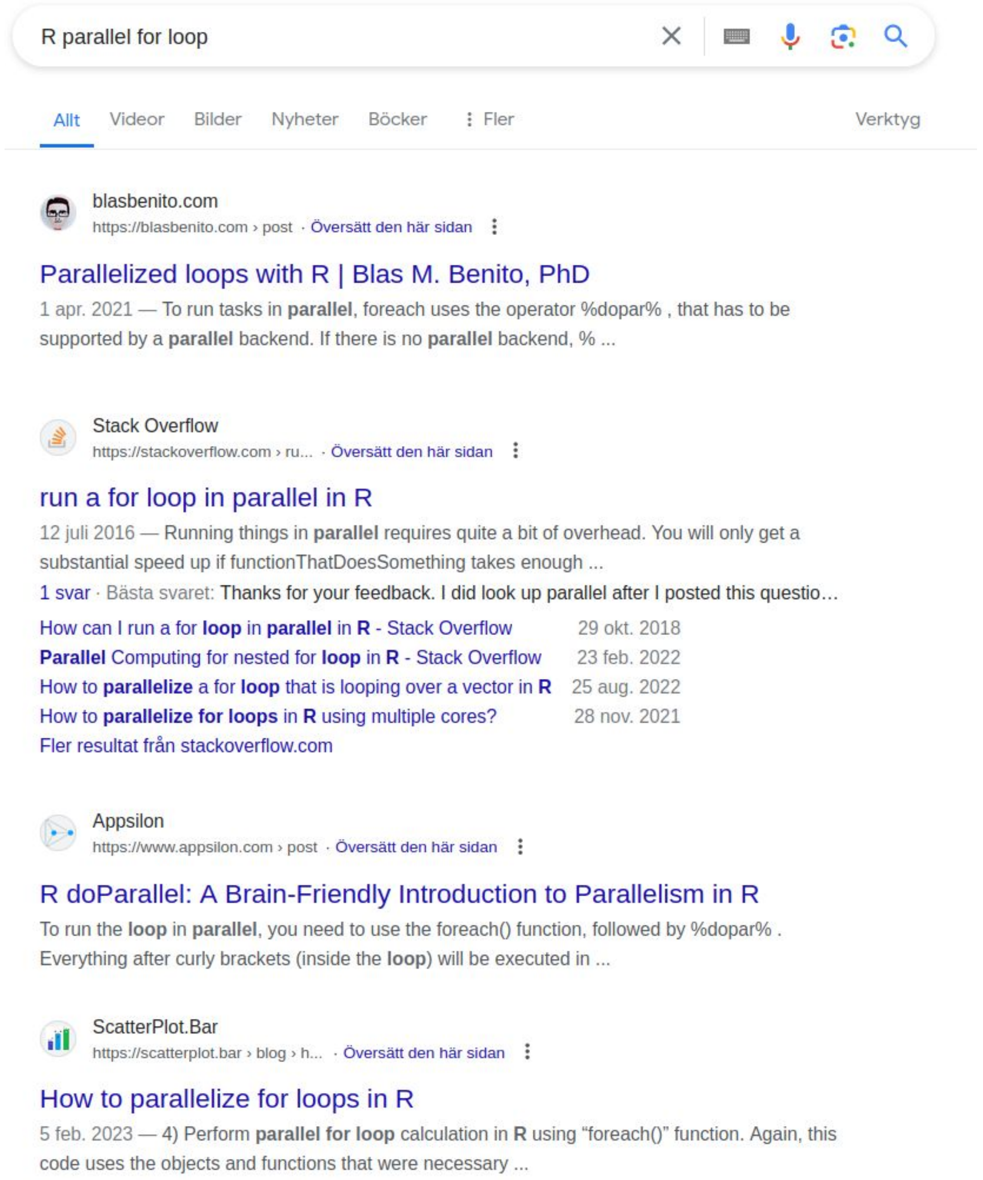
Both work equally well and are equally fast.

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!

Let’s talk foreach



R parallel for loop

blasbenito.com
https://blasbenito.com › post · Översätt den här sidan

Parallelized loops with R | Blas M. Benito, PhD

1 apr. 2021 — To run tasks in **parallel**, foreach uses the operator %dopar% , that has to be supported by a **parallel** backend. If there is no **parallel** backend, % ...

Stack Overflow
https://stackoverflow.com › ru... · Översätt den här sidan

run a for loop in parallel in R

12 juli 2016 — Running things in **parallel** requires quite a bit of overhead. You will only get a substantial speed up if functionThatDoesSomething takes enough ...

1 svar · Bästa svaret: Thanks for your feedback. I did look up parallel after I posted this questio...

How can I run a for **loop** in **parallel** in **R** - Stack Overflow	29 okt. 2018
Parallel Computing for nested for **loop** in **R** - Stack Overflow	23 feb. 2022
How to **parallelize** a for **loop** that is looping over a vector in **R**	25 aug. 2022
How to **parallelize for loops** in **R** using multiple cores?	28 nov. 2021

Fler resultat från stackoverflow.com

Appsilon
https://www.appsilon.com › post · Översätt den här sidan

R doParallel: A Brain-Friendly Introduction to Parallelism in R

To run the **loop** in **parallel**, you need to use the foreach() function, followed by %dopar% . Everything after curly brackets (inside the **loop**) will be executed in ...

ScatterPlot.Bar
https://scatterplot.bar › blog › h... · Översätt den här sidan

How to parallelize for loops in R

5 feb. 2023 — 4) Perform **parallel for loop** calculation in **R** using "foreach()" function. Again, this code uses the objects and functions that were necessary ...

Example with Duck Duck Go ...

The name foreach() tricks us to believe it's a for-loop ...

```
fq <- fs::dir_ls(glob = "*.fq")
bam <- list()
for (ii in seq_along(fq)) {
  bam[[ii]] <- align(fq[[ii]])
}
```

… but we *must never* think of it as a for-loop

```
library(foreach)

fq <- fs::dir_ls(glob = "*.fq")
bam <- list()
foreach(ii = seq_along(fq)) %dopar% {
  bam[[ii]] <- align(fq[[ii]])
}
```

This does not work because:

`foreach()` is not a for-loop!

Repeat after me: foreach() is not a for-loop!

```r
for (ii in 1:1000) {
  message("foreach() is not a for-loop!")
}
```

```
foreach() is not a for-loop!

foreach() is not a for-loop!

foreach() is not a for-loop!

foreach() is not a for-loop!

foreach() is not a for-loop!

foreach() is not a for-loop!
```

foreach() is just like lapply() ...

```
fq <- fs::dir_ls(glob = "*.fq")
bam <- list()
lapply(seq_along(fq), function(ii) {
  bam[[ii]] <- align(fq[[ii]])
})
```

This does not work, because the <- assignment is done inside a function

… and just like map() …

```
fq <- fs::dir_ls(glob = "*.fq")
bam <- list()
purrr::map(seq_along(fq), function(ii) {
  bam[[ii]] <- align(fq[[ii]])
})
```

This does not work, because the <- assignment is done inside a function

If foreach() had looked like ...

```
fq <- fs::dir_ls(glob = "*.fq")
bam <- list()
foreach(seq_along(fq), function(ii) {
  bam[[ii]] <- align(fq[[ii]])
})
```

It would be clear that the <- assignment is done inside a function

lapply(), map(), foreach() return values

```r
fq <- fs::dir_ls(glob = "*.fq")

bam <- lapply(seq_along(fq), function(ii) {
  align(fq[[ii]])
})

bam <- purrr::map(seq_along(fq), function(ii) {
  align(fq[[ii]])
})

bam <- foreach(ii = seq_along(fq)) %dopar% {
  align(fq[[ii]])
}
```

lapply(), map(), foreach() return values

```
fq <- fs::dir_ls(glob = "*.fq")

bam <- lapply(fq, align)

bam <- purrr::map(fq, align)

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

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)
...
```

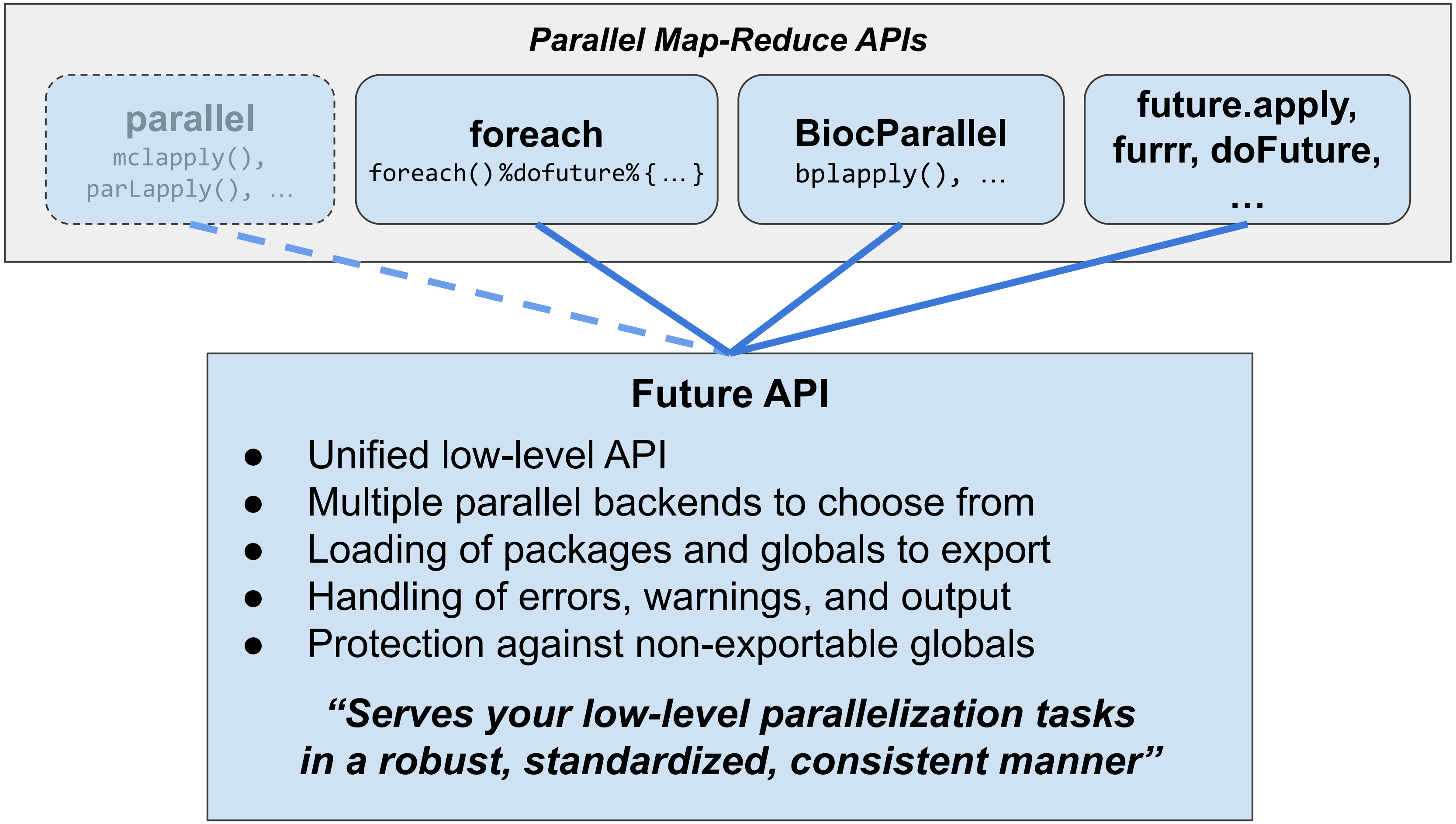
Stay with your favorite coding style 1/2

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

# Tidyverse style (purrr & furrr)
bam <- map(fq, align)
bam <- future_map(fq, 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 2/2

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

2024: Futureverse widely supported


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

Lab 2: Errors and parallel processing

- Tasks 5-9: Errors
- Tasks 10-11: Warnings

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
>
```

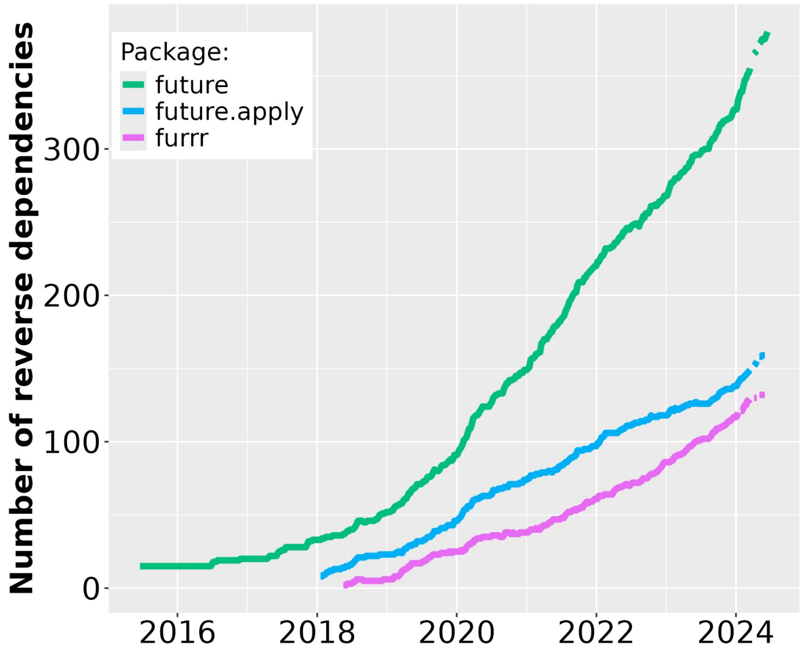
<= Output relayed from workers

<= Warnings are relayed too

Who’s using Futureverse?

Many packages use Futureverse to parallelize

- **Seurat**: Large-Scale Single-Cell Genomics
  - Instructions: "set `plan(multisession)`" from [`help("prepsctfindmarkers")`]
- **mlr3**: Next-Generation Machine Learning
  - Instructions: "set `plan(multisession)`" from [mlr3 book]



Among top-1% most installed R packages

Load balancing (“chunking”)

Chunking: All in a single round (default)

```
x <- 1:20
y <- map_dbl(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

```
plan(multisession, workers = 2)
x <- 1:20
y <- future_map_dbl(x, slow)
```

```
      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

Chunking

```
plan(multisession, workers = 3)
x <- 1:20
y <- future_map_dbl(x, slow)
```

```
       Parallel worker #1:    Parallel worker #2:    Parallel worker #3:
 1m:   y[1] <- slow(x[1])     y[8] <-  slow(x[8])    y[15] <- slow(x[15])
 2m:   y[2] <- slow(x[2])     y[9] <-  slow(x[9])    y[16] <- slow(x[16])
       …                      …                        …
 6m:   y[6] <- slow(x[6])    y[13] <- slow(x[13])    y[20] <- slow(x[20])
 7m:   y[7] <- slow(x[7])    y[14] <- slow(x[14])
```

Time: 7 mins

Chunking

```
plan(multisession, workers = 4)
x <- 1:20
y <- future_map_dbl(x, slow)
```

```
     Parallel worker #1:   Parallel worker #2:    Parallel worker #3:   Parallel worker #4:
1m:  y[1] <- slow(x[1])     y[6] <-  slow(x[6])   y[11] <- slow(x[11])  y[16] <- slow(x[16])
2m:  y[2] <- slow(x[2])     y[7] <-  slow(x[7])   y[12] <- slow(x[12])  y[17] <- slow(x[17])
     …                      …                      …                     …
5m:  y[5] <- slow(x[5])    y[10] <- slow(x[10])   y[15] <- slow(x[15])  y[20] <- slow(x[20])
```

Time: 5 mins

Uniform tasks with default chunking

10 tasks: █ █ █ █ █ █ █ █ █ █

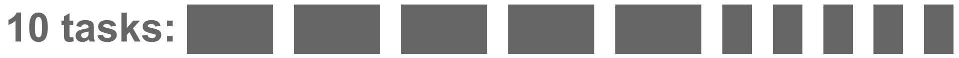
```
y <- future_map_dbl(x, slow)
```

Worker #1: █ █ █ █ █
Worker #2: █ █ █ █ █

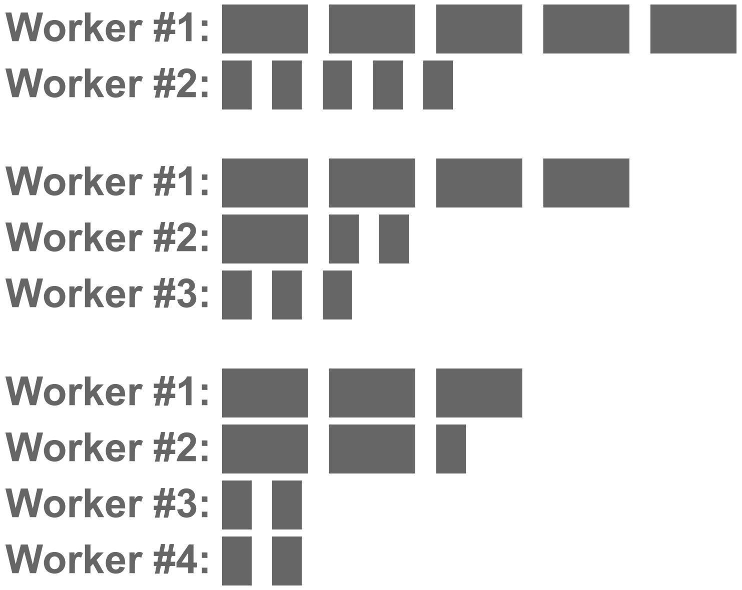
Worker #1: █ █ █ █
Worker #2: █ █ █
Worker #3: █ █ █

Worker #1: █ █ █
Worker #2: █ █ █
Worker #3: █ █
Worker #4: █ █

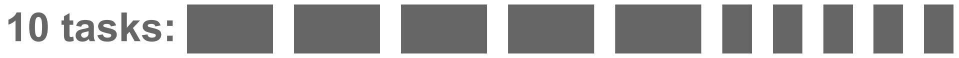
Variables tasks with default chunking



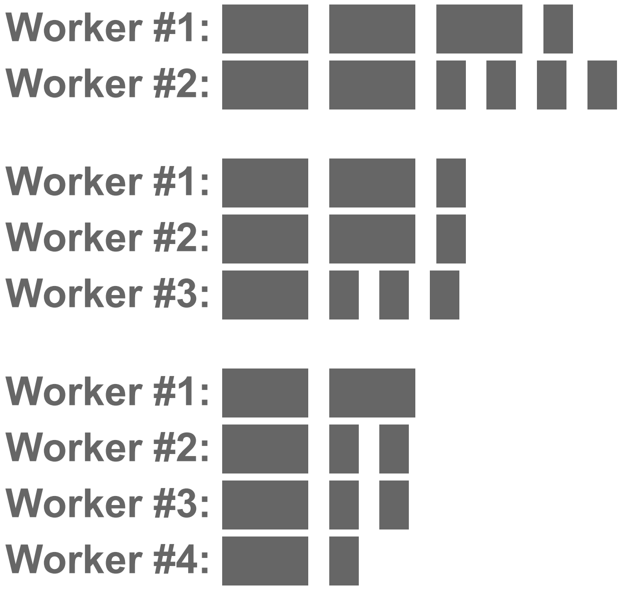
```
y <- future_map_dbl(x, slow)
```



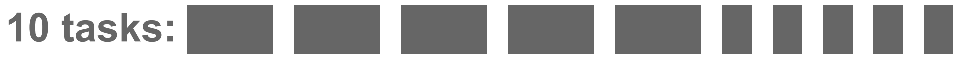
Variables tasks with teeny chunks



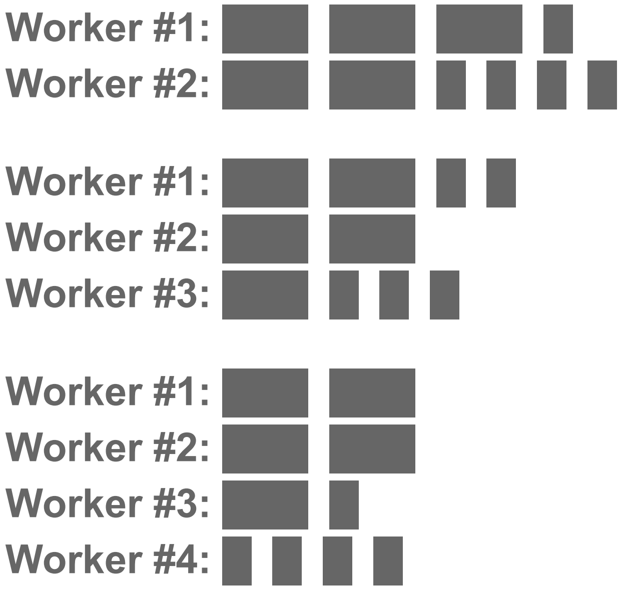
```
y <- future_map_dbl(x, slow,
                    .options=future_options(chunk.size=1))
```



Variables tasks with small chunks



```
y <- future_map_dbl(x, slow,
                    .options=future_options(chunk.size=2))
```



High Performance Compute (HPC)

Backend package: future.batchtools

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

fq <- fs::dir_ls(glob = "*.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
...
```

Progress Updates

progressr - Inclusive, Unifying API for Progress Updates

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

API for Developers:

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

<u>Developer decides:</u>

where in the code progress updates should be signaled

API for Users:

```r
handlers(global = TRUE)
handlers("cli")
```

<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 <- map_dbl(x, function(z) {
    p(paste0("z=", z))
    slow(z)
  }
  sum(y)
}
```

User

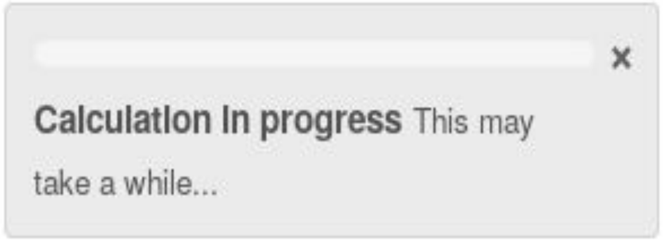
```
> handlers(global = TRUE)
> x <- 1:50
> y <- snail(x)
  [==============>--]  80% z=40
```

User decides how progress is presented

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

> handlers("beepr")
> y <- snail(x)
♬ ♪ ♪ ♪ … ♬

> handlers("cli", "beepr")
> y <- snail(x)
[======>------------]  40% z=20
♬ ♪ ♪ ♪ … ♬
```

Works also with Shiny

`withProgressShiny()`



Calculation in progress This may take a while...

future + progressr = ❤

Futureverse supports live progress updates

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

```
> handlers(global = TRUE)
> handlers("cli”, "beepr")
> plan(multisession)
> y <- snail(x)
 [===========>]  90% z=32 by 3003
```

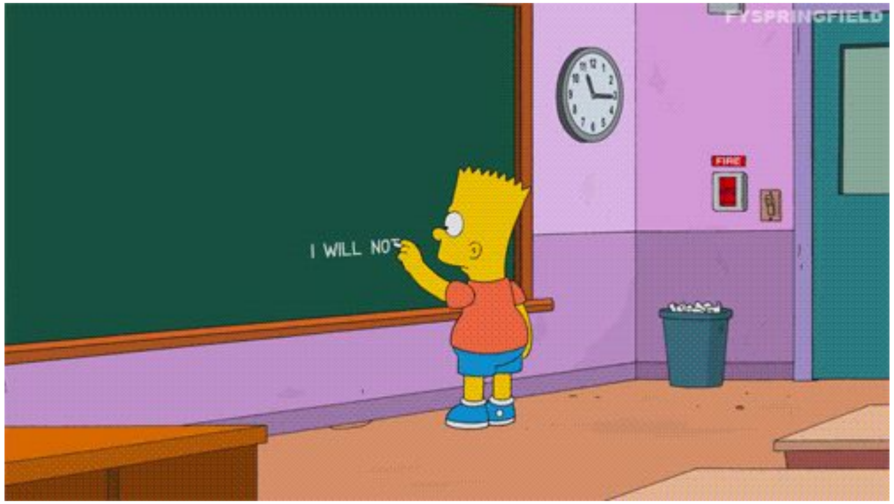
♬ ♪ ♪ ♪ … ♬

Lab 2: Progress updates

- Task 12-17: Progress updates and customization
- Task 18: Progress updates in parallel

Take home: future = 99% worry-free parallelization

- Use Futureverse instead of mclapply(), parLapply(), doParallel(), …
- Use future.apply, furrr, or foreach with doFuture - your choice
- "Write once, run anywhere" - compute clusters too
- Global variables - automatically taken care of
- Stdout, messages, warnings, *progress* - captured and relayed



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