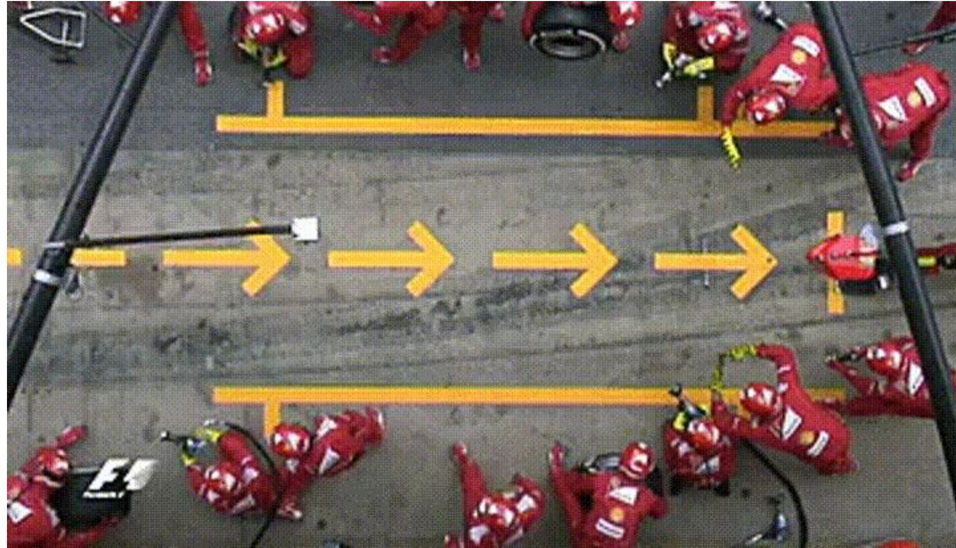
Future: A Simple, Extendable, Generic Framework for Parallel Processing in R



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

European Bioconductor Meeting 2020 on 2020-12-18 (a 35-minute presentation)

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?

R comes with built-in parallelization

```
library(DNAseq)
fq <- c("a.fq", "b.fq", "c.fq")                         # FASTQ files
bam <- lapply(fq, align)                                 # 3 hours
```

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 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 => unstable with *multi-threaded* code & GUIs, e.g. may core dump RStudio

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"))`

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

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

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

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

v3. An alternative approach

```
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?

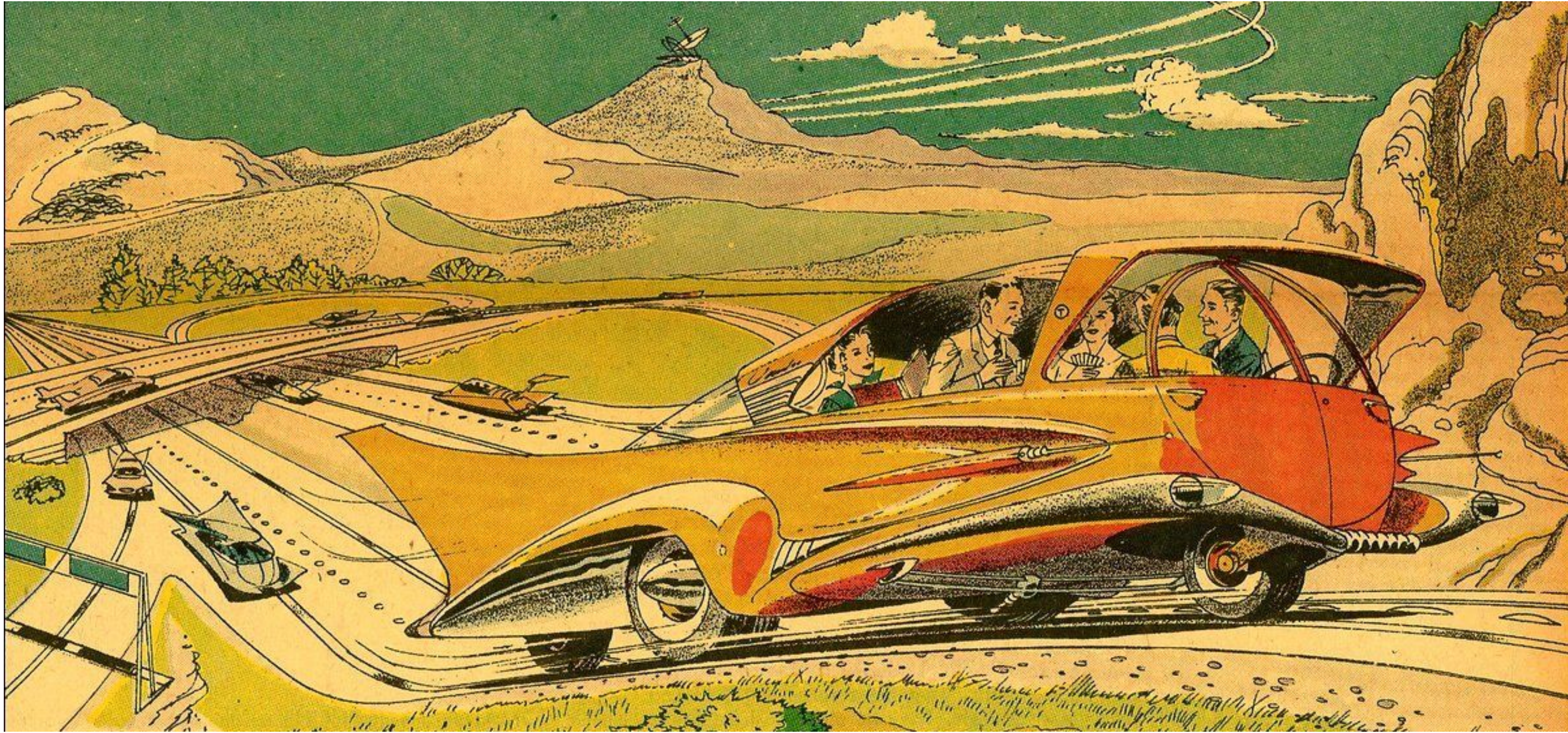
SOLUTION: Parallelization frameworks

Quoting BiocParallel:

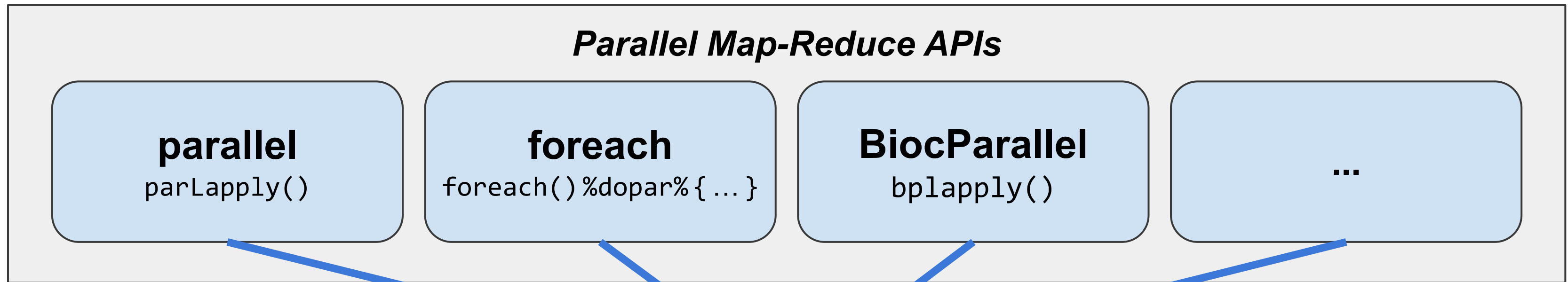
> *"A basic objective … is to reduce the complexity faced when developing and using software that performs parallel computations. … aims to provide a unified interface to existing parallel infrastructure where code can be easily executed in different environments"*

Source: BiocParallel vignette 'Introduction to BiocParallel'

Welcome to the Future



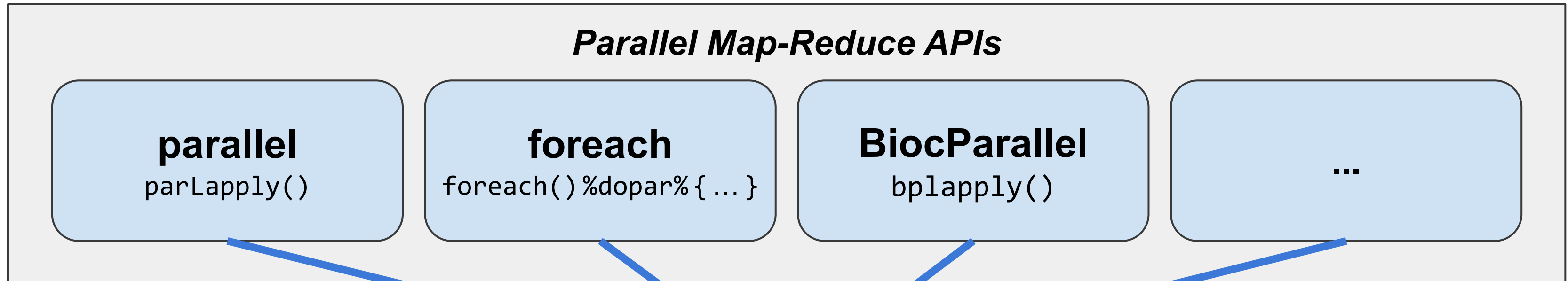
Parallel frameworks reimplement common ideas



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



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"

R package: future

Dan LaBar
@embiggenData

- "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)

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

```r
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( { align(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)
}
```

```r
> 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)
...
```

Backend package: future.batchtools

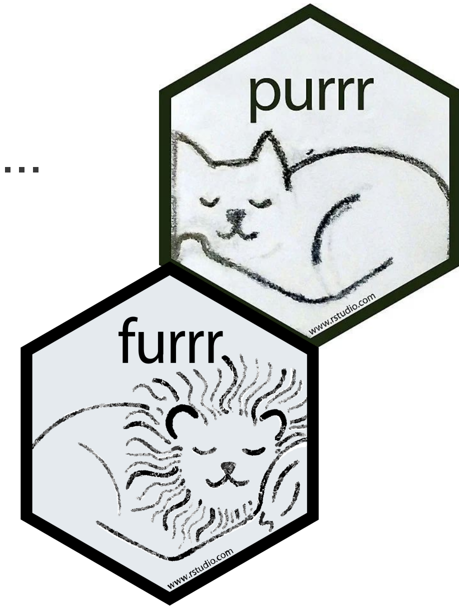
```
plan(future.batchtools::batchtools_slurm)

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       01:32:05 R
606641   python             bob          37:18:30 R
606643   future_lapply-6    henrik       01:31:55 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




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

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

Package: doFuture

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

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

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

Stay with your favorite coding style

```
# 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) %dopar% align(x)
```

Also BiocParallel

```
library(BiocParallel)
register(DoparParam())   # BiocParallel to use %dopar%
doFuture::registerDoFuture()  # %dopar% to use futures
future::plan("multisession")

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

Use BiocParallel with futures because …

- Consistent behavior regardless of parallel backend
- Standard output is truly relayed
- Messages, warnings, and other conditions are relayed as-is
- Optional protection against non-exportable globals
- Parallel near-live progress updates via **progressr** framework

Output, Warnings, and Errors

Output and warnings behave consistently for all parallel backends

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

Whether or not output is visible depends on operating system and environment (e.g. terminal or RStudio)

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 relayed from workers

<= Warnings are relayed too

Same problems with BiocParallel

```
> register(SnowParam(2))
> stdout <- capture.output({
    y <- bplapply(x, function(z) {
      str(z)
      log(z)
    })
  })
 num -1
 num 10
 num 30
Warning message:
In FUN(X[[i]], ...) : NaNs produced
> stdout
[1] character(0)
```

<= This is *not* outputted in the R session but in the underlying terminal

<= Same for this warning ...

<= So, nothing is captured

But, with a little help from a friend: standard output is truly relayed

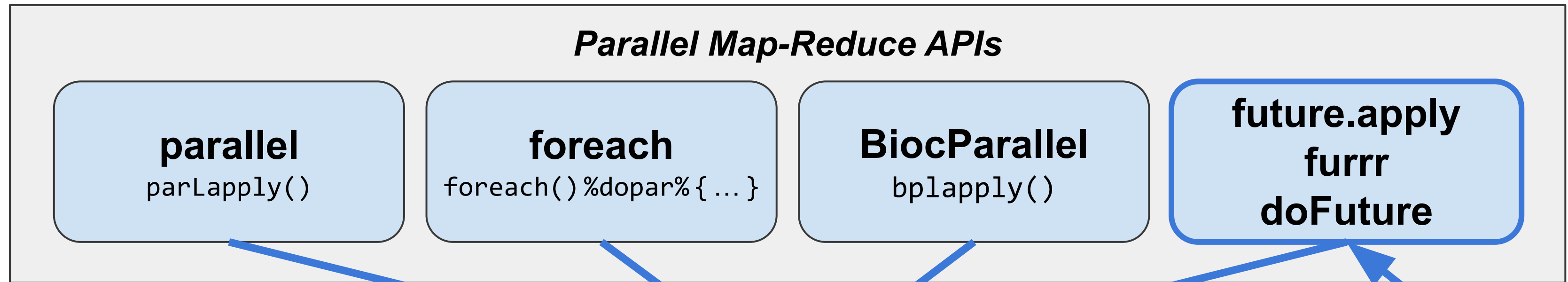
```
> register(DoparParam()); registerDoFuture()
> plan(multisession, workers = 2)

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

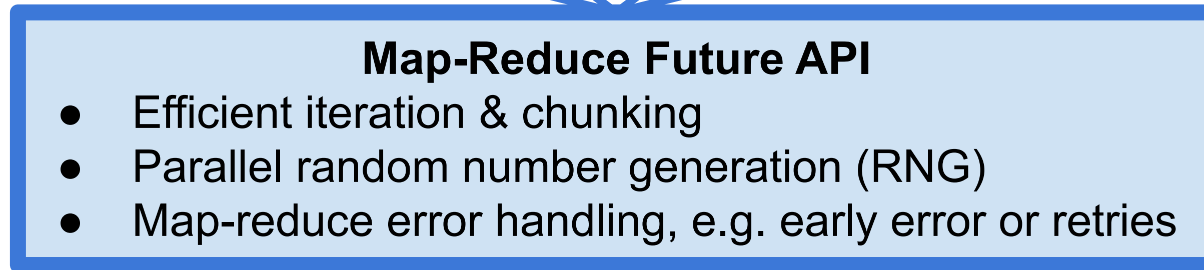
When plugging in the future framework, output is captured and warnings are relayed 👍

Just like for lapply() or with register(SerialParam()) 👍

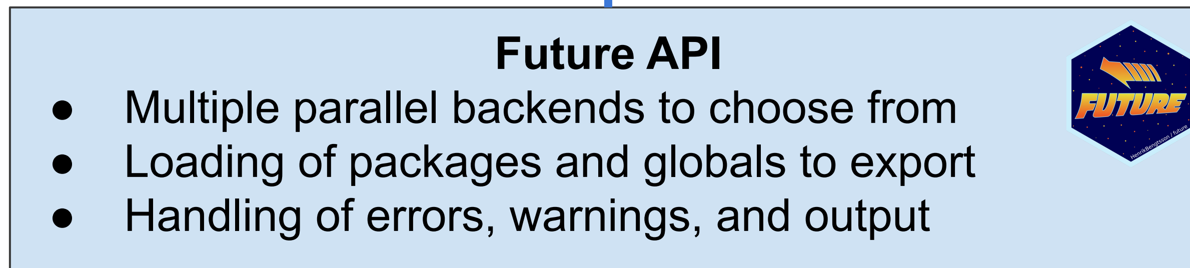
Map-Reduce Future API on top of Future API



in the works

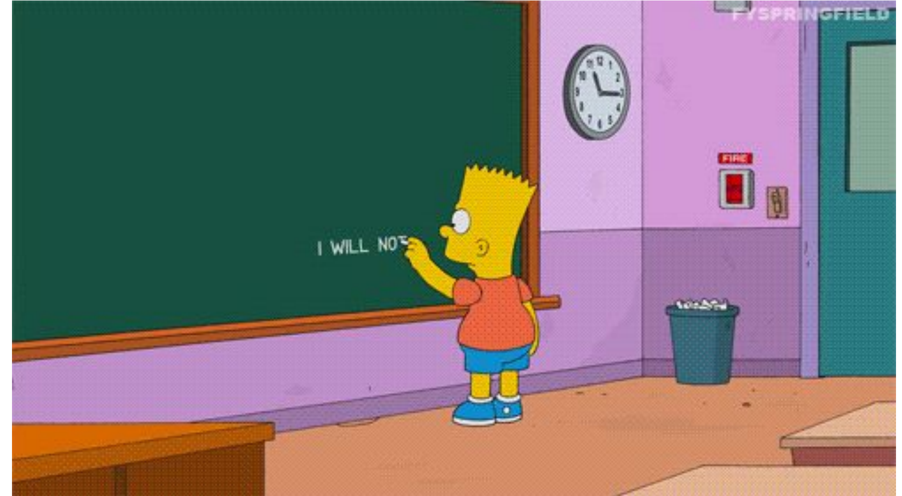


already in these packages



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



@HenrikBengtsson

HenrikBengtsson

jottr.org

Q & A

Q. Profile memory usage

Q: *planning a feasible memory footprint seems important ... do you have examples on how to "instrument" usage of this framework. [Follow-up clarification:] measure the impact of choosing the number of cores, say ... as you add more jobs to run in parallel you will eventually exhaust available ram. do you have ways of helping users to assess this risk? sounds like your "resources" discussion is relevant.*

A: It's on the roadmap to collect benchmark information on futures to help profile performance - time and memory. The simplest will be to collect timing information throughout the lifespan of a future, e.g. time to create the future, time to find an available worker, time to export objects to the worker, time for worker evaluate the future expression, time to send back the results to the parent R process, and time to relay standard output and conditions.

Currently, the only information collected is the start and finish timestamps of the evaluation. There is no API for exposing this right now, i.e. this information is currently only available as internal fields. For example,

Q. Profile memory usage (continued)

```
> f <- future({ Sys.sleep(2.0); 42 })
> v <- value(f)
```

We can also ask for the detailed "results", which contain information such as captured output and conditions, among other things. (This structure is currently not part of the official API and may change at any time.) Here is how we can see how long it took for the worker to resolve the future expression:

```
> r <- result(f)
> r$finished - r$started
Time difference of 2.004035 secs
```

Gather information on memory use could work similarly. However, as a mentioned in my online answer, collecting how memory is in use can be really complicated but there are a few third-party packages that provide solutions for this.

Q. Profile memory usage (continued)

There are three main hurdles to be resolved in order to implement profiling:

1. Ideally, it should be possible to plug-in custom benchmarking tools. This is probably best done via hook functions. Support for hook functions is also on the roadmap.

2. Benchmarking, and hook functions, will probably introduce additional overhead. Because of this, it should be optional and when not in use the overhead should be near-zero or ideal truly zero.

3. How these benchmark information should be accessed needs to be identified.

This is tracked in https://github.com/HenrikBengtsson/future/issues/59.

Q. Send a future to a specific worker?

Q: *Can we manually assign the computations over several workers/manager, to make sure the computations won't have conflicts (Linux and Mac)?*

A: I'm not sure I understand the questions but I'll make an attempt to answer what I think is asked. The Future API does not expose the concept of a "worker". For example, we cannot do:

```
f <- future(sum(x), worker = “machine5”)
```

That would against the philosophy that a future should be able to be processed anywhere, e.g. on the local machine, on another machine, via a job scheduler, or in the cloud. However, there are plans to support specifying "resource" requirements, e.g.

```
f <- future(sum(x), resource = c(os=“linux”, mem=”50gb“))
```

Only a worker that can support these needs, will take on the future.

Q. Custom ways to combine results?

Q: do the APIs provide also a way to combine the results of the parallelized computations? (i.e. cbind/rbind/etc) or maybe providing a user defined function for combining the results

A: By design, the Future API provides only three functions: **`f <- future(expr)`**, **`r <- resolved(f)`**, and **`v <- value(f)`**. The latter two functions are generic functions with S3 methods not only single futures, but also futures in lists and environments. For instance, instead of using **`lapply(fs, value)`** as on Slide 26, we can do:

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

Other way to combine will be your favorite R functions, or whatever the higher-level, parallel map-reduce APIs such as those **future.apply**, **furrr**, **foreach**, **BiocParallel** provide.