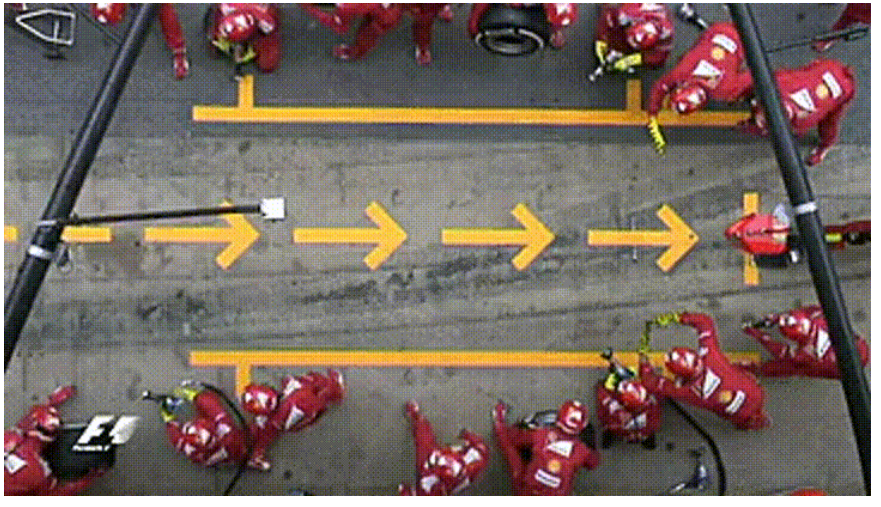
Future: Friendly Parallel Processing in R for Everyone

Henrik Bengtsson
Univ of California, San Francisco

@HenrikBengtsson
HenrikBengtsson/future
jottr.org



Acknowledgments

- SatRday Paris 2019
- AgroParisTech
- R Core, CRAN, devels, and users!
- R Consortium

A 40-minute presentation, SatRday Paris 2019, Paris, 2019-02-23

Why do we parallelize software?

Parallel & distributed processing can be used to:

1. **speed up processing** (wall time)
2. **lower memory footprint** (per machine)
3. **avoid data transfers** (compute where data lives)
4. Other reasons, e.g. asynchronous UI/UX

How do we parallelize in base R?

since R 2.14.0 (Nov 2011)

```
X <- list(a = 1:50, b = 51:100, c = 101:150, d = 151:200)
y <- lapply(X, FUN = slow_sum)   # 4 minutes
```

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

```
y <- parallel::mclapply(X, FUN = slow_sum, mc.cores = 4)   # 1 minute
```

To parallelize also on Windows, we can do:

```
library(parallel)
workers <- makeCluster(4)

clusterExport(workers, "slow_sum")
y <- parLapply(workers, X, fun = slow_sum)   # 1 minute
```

PROBLEM: Different APIs for different parallelization strategies

Developer

- "Which parallel API should I use?"
- "What operating systems are users running?"
- "It should work ... Oh, I forgot to test on macOS."

User

- "Weird, others say it work for them but for me it doesn't!?"
- "I wish this awesome package could parallelize on Windows :("
- "I wish we could use a compute cluster in the cloud to speed this up"

PROBLEM: Code clutter + error prone

```
#' @import parallel
my_fun <- function(X, ncores = 1) {
  if (ncores == 1) {
    y <- lapply(X, FUN = my_sum)
  } else {
    if (.Platform$OS.type == "windows") {
      workers <- makeCluster(ncores)
      on.exit(stopWorkers(workers))
      clusterExport(workers, "slow_sum")
      y <- parLapply(workers, X, fun = slow_sum)
    } else {
      y <- mclapply(X, FUN = my_sum, mc.cores = ncores)
    }
  }
  y
}
```

SOLUTION: Encapsulate these problems

```
library(foreach)
doMC::registerDoMC(4)  # User chooses how to parallelize

my_fun <- function(X) {
  foreach(x = X) %dopar% { slow_sum(x) }
}
```

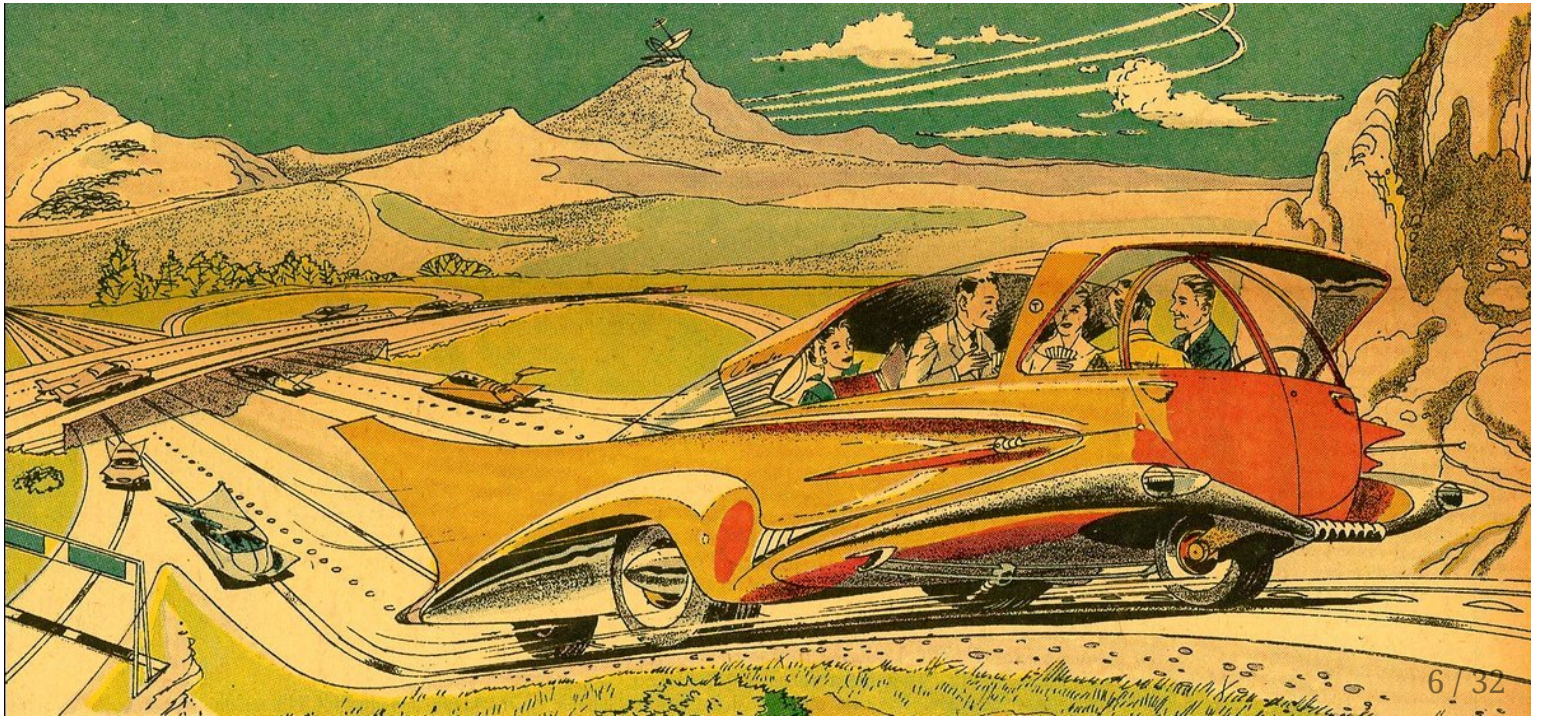
```
workers <- parallel::makeCluster(4)  # Alternative parallel backend
doParallel::registerDoParallel(workers)
```

```
Error in { : task 1 failed - 'could not find function "slow_sum"'
```

Whoops, we forgot to export **`slow_sum()`** to background sessions;

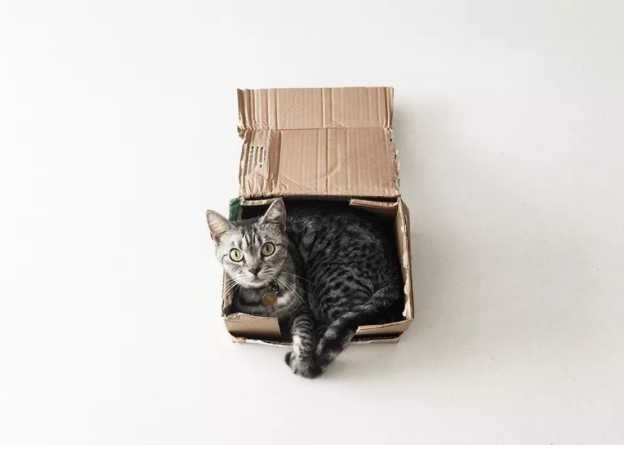
```
  foreach(x = X, .export = "slow_sum") %dopar% { slow_sum(x) }
```

The Future ...



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 **unevaluated** or **evaluated**



Friedman & Wise (1976, 1977), Hibbard (1976), Baker & Hewitt (1977), … Schrödinger (1935)?

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 **unevaluated** or **evaluated**

Standard R:

```
v <- expr
```

Future API:

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

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)

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

> value(fa)
[1] 1275
> value(fb)
[1] 3775

> value(fa) + value(fb)
[1] 5050
```

Two alternative syntaxes

Standard R:

```
v <- expr
```

Future API (explicit):

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

Future API (implicit):

```
v %<-% expr
```

Example: Sum of 1:50 and 51:100 in parallel (implicit API)

```
> library(future)
> plan(multiprocess)

> a %<-% slow_sum( 1:50 )   # ~0 seconds
> b %<-% slow_sum(51:100)   # ~0 seconds
> 1:3
[1] 1 2 3

> a + b
[1] 5050
```

Many ways to resolve futures

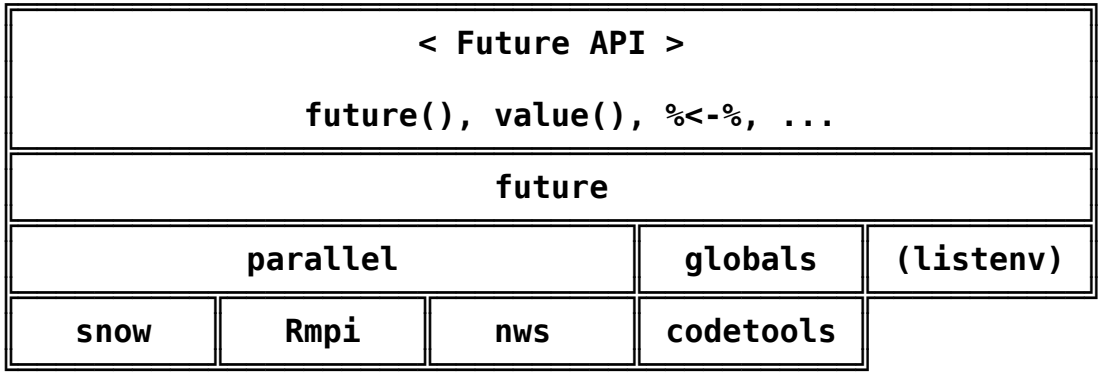
```
plan(sequential)
plan(multiprocess)
plan(cluster, workers = c("n1", "n2", "n3"))
plan(cluster, workers = c("remote1.org", "remote2.org"))
...
```

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

R package: future

CRAN 1.11.1.1

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



Why a Future API? Solution: "interface of interfaces"

- The Future API encapsulates heterogeneity
  - fever decisions for developer to make
  - more power to the end user
- Motto: **Developer decides what to parallelize** - **user decides how to**
- Provides **atomic building blocks**:
  - **`f <- future(expr)`**, **`v <- value(f)`**, ...

  for richer parallel constructs, e.g. 'foreach', 'future.apply', ...
- **Automatic support for new backends**,
  e.g. 'future.callr', 'future.batchtools', 'future.clustermq', ...

Why a Future API? 99% Worry Free

- **Globals & Packages**: automatically **identified & exported**
- **Static-code inspection** by walking the abstract syntax tree (AST)

```
x <- rnorm(n = 100)
f <- future({ slow_sum(x) })
```

```
## pryr::ast(  { slow_sum(x) }  )
## \- `{
## \- ()
##   \- `slow_sum
##   \- `x
```

Globals identified and exported to background R worker:

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

Globals & packages can also be manually specified

Building things using core future building blocks

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

Building things using core future building blocks

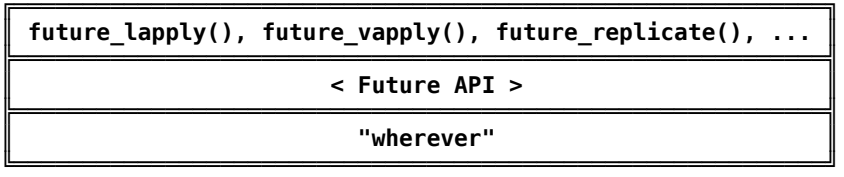
```
#' @import future
parallel_lapply <- function(X, fun, ...) {
  ## Create futures
  fs <- lapply(X, function(x) {
    future(fun(x, ...))
  })

  ## Collect their values
  lapply(fs, value)
}
```

```
> plan(multiprocess)
> X <- list(a = 1:50, b = 51:100, c = 101:150, d = 151:200)
> y <- parallel_lapply(X, slow_sum)
> str(y)
List of 4
 $ a: int 1275
 $ b: int 3775
 $ c: int 6275
 $ d: int 8775
```

Frontend: future.apply

CRAN 1.1.0

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

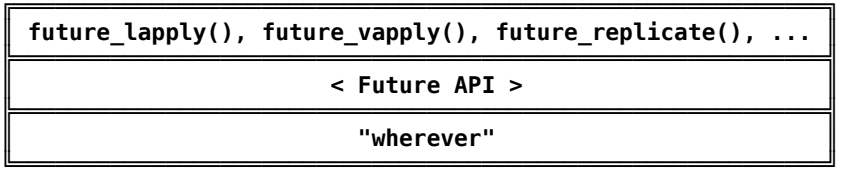



```
y <- lapply(X, slow_sum)
```

Frontend: future.apply

CRAN 1.1.0

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

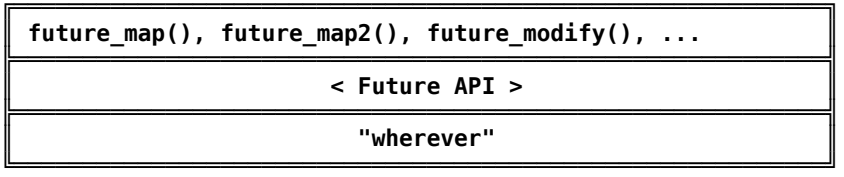



```
y <- future_lapply(X, slow_sum)
```

- **`plan(multiprocess)`**
- **`plan(cluster, workers = c("n1", "n2", "n3"))`**
- **`plan(batchtools_sge)`**

Frontend: furrr (Davis Vaughan)

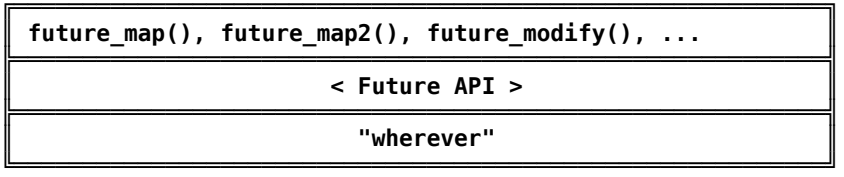
- Futurized version of purrrs R's **`map()`**, **`map2()`**, **`modify()`**, ...
- ... on **all future-compatible backends**




```
y <- purrr::map(X, slow_sum)
```

Frontend: furrr (Davis Vaughan)

- Futurized version of purrrs R's **`map()`**, **`map2()`**, **`modify()`**, ...
- ... on **all future-compatible backends**

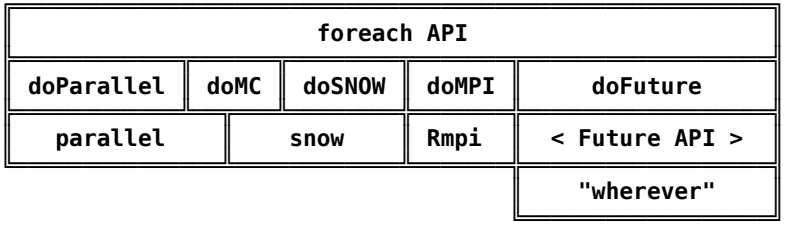



```
y <- future_map(X, slow_sum)
```

Frontend: doFuture

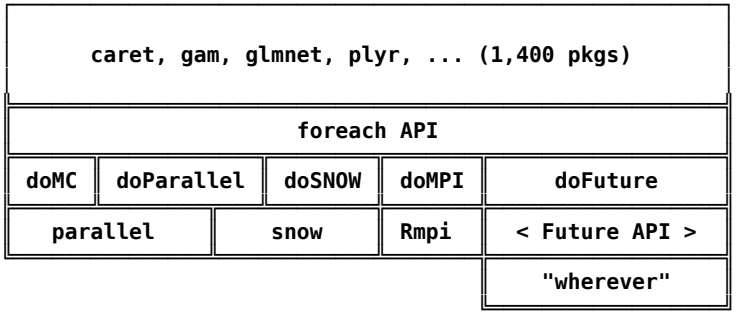
CRAN 0.7.0

- A **foreach** adapter on top of the Future API
- Foreach on **all future-compatible backends**



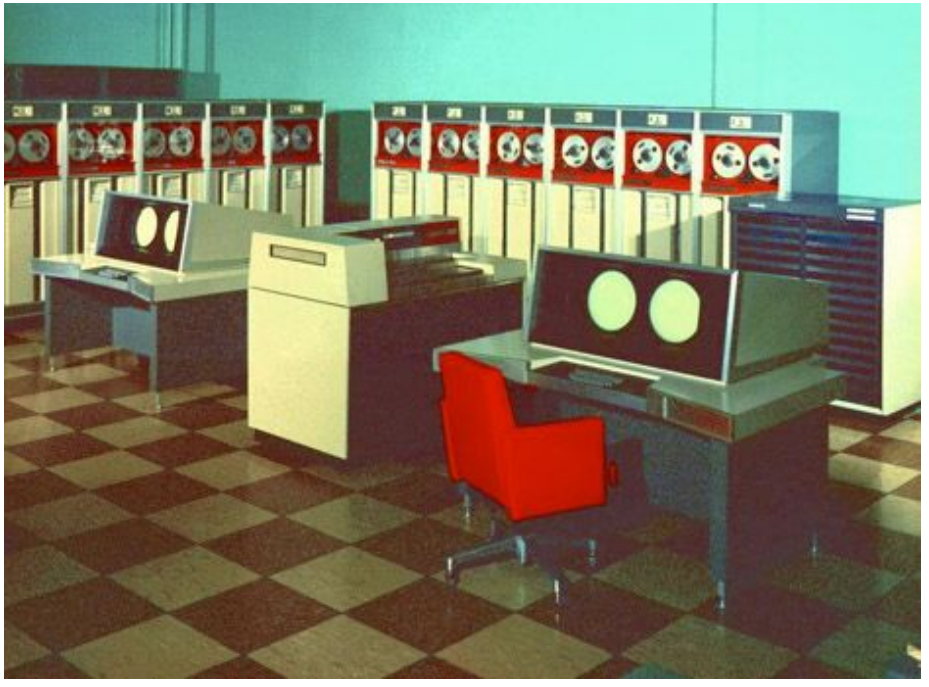

```
doFuture::registerDoFuture()
plan(batchtools_sge)
y <- foreach(x = X) %dopar% { slow_sum(x) }
```

~1,400 packages can now parallelize on HPC




```
doFuture::registerDoFuture()
plan(future.batchtools::batchtools_sge)

library(caret)
model <- train(y ~ ., data = training)
```

High Performance Compute (HPC) clusters



Example: Genome sequencing project

- Sequencing of a human DNA ($3 * 10^9$ nucleotides)
- 80 individuals
- Millions of short raw sequences need to be mapped to the human reference
- Alignment takes ~3 hours per individual
- Raw sequence data is ~200 GB per individual

```
## Find our 80 FASTQ files
fastq <- dir(pattern = "[.]fq$")          ## 200 GB each => 16 TB total

## Align the to human genome
bam <- lapply(fastq, DNAseq::align)       ## 3 hours each
```

Total processing time: 80 * 3 = 240 hours = 10 days

Example: Genome sequencing project

- Sequencing of a human DNA ($3 * 10^9$ nucleotides)
- 80 individuals
- Millions of short raw sequences need to be mapped to the human reference
- Alignment takes ~3 hours per individual
- Raw sequence data is ~200 GB per individual

```
library(future.apply)
plan(multiprocess)     ## 12-core machine

## Find our 80 FASTQ files
fastq <- dir(pattern = "[.]fq$")              ## 200 GB each => 16 TB total

## Align the to human genome
bam <- future_lapply(fastq, DNAseq::align)    ## 3 hours each
```

Total processing time: 80 * 3 / 12 = 20 hours

Ad-Hoc Compute Clusters

A common setup in many departments:

- Two or more machines
- Manually SSH into each machine to launch scripts

Attributes:

- Works ok with a few people and fair usage
- Can easily be overloaded if too many users
- Hard to plan your jobs

Clusters with Job Queues

With too many nodes or users, ad-hoc clusters becomes cumbersome and hard to manage and control. **Better to use a HPC scheduler with a job queue**:

- Two or more machines
- Users submit jobs to a common job queue
- The system takes jobs on the queue and executes them on available machines / cores

Attributes:

- Works well with any number of users and machines
- Users do not have to worry about overloading the cluster; the cluster will wait to process the next job if all compute resources are busy running jobs

Example: Submit a job & watch the queue

```
#! /bin/env bash
#PBS -N my_htseq_align
#PBS -l mem=12gb

htseq_align $1 human.fa
```

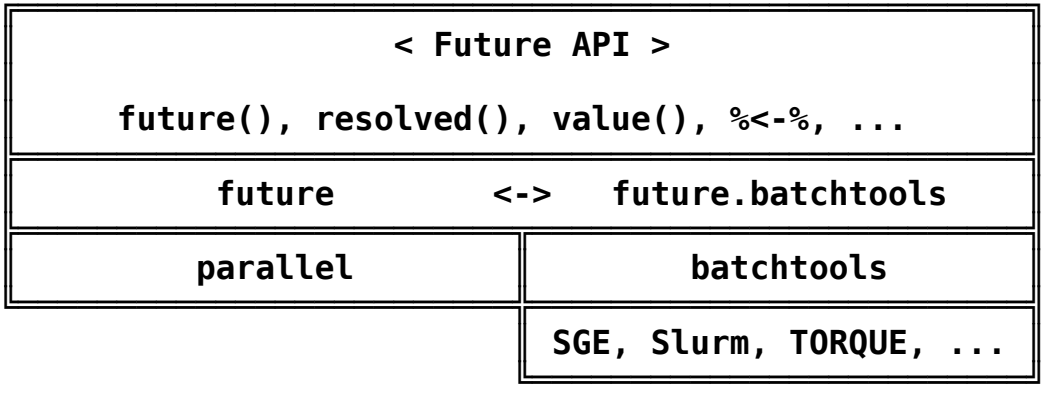
```
$ qsub htseq_align.pbs patient101.fq
$ qsub htseq_align.pbs patient102.fq
```

```
$ qstat
Job ID   Name             User         Time Use S
-------- ---------------- ------------ -------- -
606411   bedGraph         alice        46:22:22 R
606494   misosummary      alice        55:07:08 R
606641   Rscript          bob          37:18:30 R
607758   Exome_QS1_Som    charlie      06:20:23 R
607832   my_htseq_align   henrik       00:01:57 R
607833   my_htseq_align   henrik              - Q
```

Backend: future.batchtools

CRAN 0.7.2

- **batchtools**: Map-Reduce API for **HPC schedulers**,
  e.g. LSF, OpenLava, SGE, Slurm, and TORQUE / PBS
- **future.batchtools**: **Future API** on top of **batchtools**



Backend: future.batchtools


```
library(future.batchtools)
plan(batchtools_sge)

fastq <- dir(pattern = "[.]fq$")              ## 200 GB each; 80 files
bam <- future_lapply(fastq, DNAseq::align)    ## 3 hours each
```

```
$ qstat
Job ID   Name             User         Time Use S
-------- ---------------- ------------ -------- -
606411   bedGraph         alice        46:22:22 R
606638   future05         henrik       01:32:05 R
606641   Rscript          bob          37:18:30 R
606643   future06         henrik       01:31:55 R
...
```

Backend: Google Cloud Engine Cluster (Mark Edmondson)

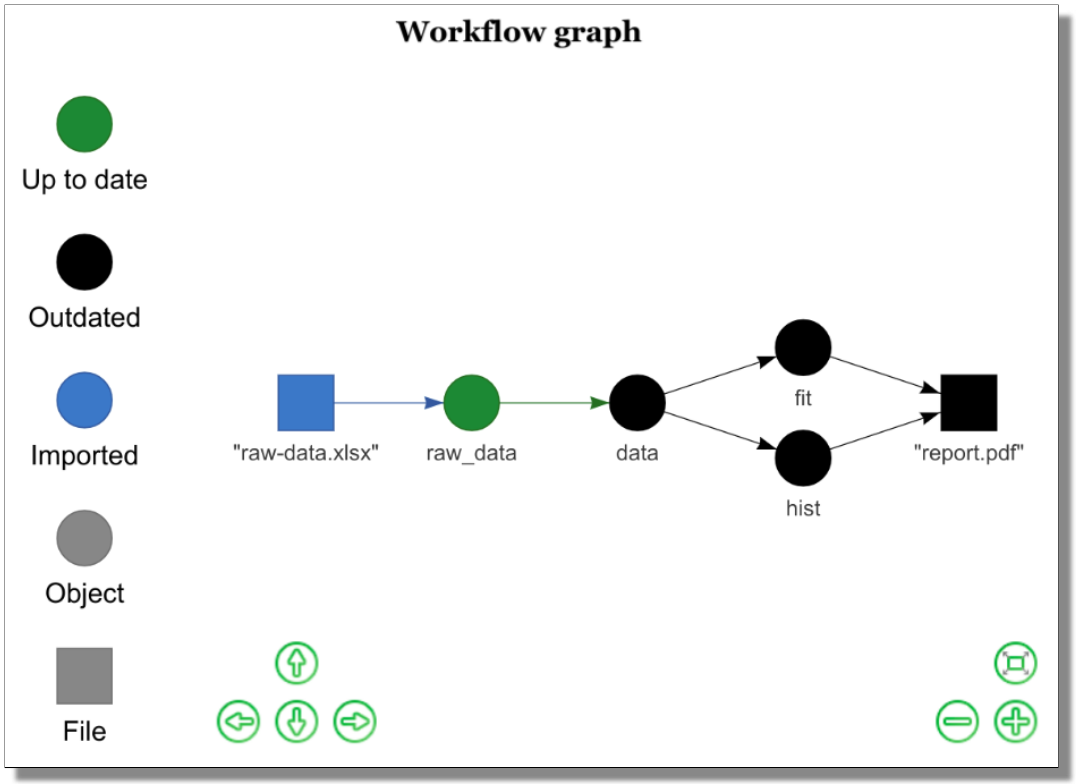
```
library(googleComputeEngineR)
vms <- lapply(paste0("node", 1:10),
              FUN = gce_vm, template = "r-base")
cl <- as.cluster(lapply(vms, FUN = gce_ssh_setup),
                 docker_image = "henrikbengtsson/r-parallel")

plan(cluster, workers = cl)
```

```
data <- future_lapply(1:100, montecarlo_pi, B = 10e3)
pi_hat <- Reduce(calculate_pi, data)

print(pi_hat)
## 3.14159
```

Futures in the Wild ...

drake - A Workflow Manager (Will Landau & rOpenSci)


```
tasks <- drake_plan(
  raw_data = readxl::read_xlsx(file_in("raw-data.xlsx")),

  data = raw_data %>% mutate(Species =
         forcats::fct_inorder(Species)) %>% select(-X__1),

  hist = ggplot(data, aes(x = Petal.Width, fill = Species))
         + geom_histogram(),

  fit = lm(Sepal.Width ~ Petal.Width + Species, data),

  rmarkdown::render(knitr_in("report.Rmd"),
                    output_file = file_out("report.pdf"))
)

future::plan("multiprocess")
make(tasks, parallelism = "future")
```


Workflow graph
Up to date
Outdated
Imported
Object
File
"raw-data.xlsx"
raw_data
data
fit
hist
"report.pdf"

shiny - Asynchronous UI (RStudio)

> Shiny v1.1 (the one with async) is days away from release! Huge changes under the hood--it'd be a big help if you try out your app using devtools::install_github("rstudio/shiny") and let us know if anything breaks! #rstats
>
>  **Joe Cheng** @jcheng
> 6:40pm - 11 May 2018

```
library(shiny)
future::plan("multiprocess")
...
```

The Near Future ...

Improvements

- ☑ Capture and relay standard output
- ☑ Capture and relay messages and warnings
- ☑ Improved error reporting and tracebacks (more can be done)
- ☐ [50%] Benchmarking (time and memory)
- ☐ [20%] Hook functions, e.g. update progress bar when one future is resolved
- ☐ [20%] Killing futures
- ☐ [10%] Restarting failed futures
- ☐ [80%] **future.tests** - Unified test framework for all future backends
  - All parallel backends must support the Core Future API
  - Sponsored via an R Consortium grant

foreach

- ☐ [80%] HARMONIZATION: Identify globals using **future**

Roadmap

- Make future **truly pure "containers"** to be evaluated
  - One of the original design goals
  - Support for passing futures "asis" to wherever and whenever
- Specify **resource needs**, e.g. only pass future to a worker:
  - ... on the same file system
  - ... that has a GPU
- **Sandboxed** future backend, e.g. evaluate non-trusted code

Summary of features

- **Unified API**
- **Portable code**
- **Worry-free**
- **Developer decides what to parallelize - user decides how to**
- For beginners as well as advanced users
- Nested parallelism on nested heterogeneous backends
- Protects against recursive parallelism
- Easy to build new frontends
- Easy to add new backends

Building a better future

I 💜 feedback,
bug reports,
and suggestions

@HenrikBengtsson

HenrikBengtsson/future

jottr.org

Thank you!

Appendix (Random Slides)

A1. Features - more details

A1.1 Well Tested

- Large number of unit tests
- System tests
- High code coverage (union of all platform near 100%)
- Cross platform testing
- CI testing
- Testing several R versions (many generations back)
- Reverse package dependency tests
- All backends highly tested
- Large of tests via doFuture across backends on **`example()`**:s from foreach, NMF, TSP, glmnet, plyr, caret, etc. (example link)

R Consortium Infrastructure Steering Committee (ISC) Support Project

- **Backend Conformance Test Suite** - an effort to formalizing and standardizing the above tests into a unified go-to test environment.

A1.2 Nested futures

```
fastq <- dir(pattern = "[.]fq$")

aligned <- listenv()
for (i in seq_along(fastq)) {
  aligned[[i]] %<-% {
    chrs <- listenv()
    for (j in 1:24) {
      chrs[[j]] %<-% DNAseq::align(fastq[i], chr = j)
    }
    merge_chromosomes(chrs)
  }
}
```

- **plan(batchtools_sge)**
- **plan(list(batchtools_sge, sequential))**
- **plan(list(batchtools_sge, multiprocess))**

A1.3 Lazy evaluation

By default all futures are resolved using eager evaluation, but the *developer* has the option to use lazy evaluation.

Explicit API:

```
f <- future(..., lazy = TRUE)
v <- value(f)
```

Implicit API:

```
v %<-% { ... } %lazy% TRUE
```

A1.4 False-negative & false-positive globals

Identification of globals from static-code inspection has limitations (but defaults cover a large number of use cases):

- False negatives, e.g. **`my_fcn`** is not found in **`do.call("my_fcn", x)`**. Avoid by using **`do.call(my_fcn, x)`**.
- False positives - non-existing variables, e.g. NSE and variables in formulas. Ignore and leave it to run-time.

```
x <- "this FP will be exported"

data <- data.frame(x = rnorm(1000), y = rnorm(1000))

fit %<-% lm(x ~ y, data = data)
```

Comment: ... so, the above works.

A1.5 Full control of globals (explicit API)

Automatic (default):

```
x <- rnorm(n = 100)
y <- future({ slow_sum(x) }, globals = TRUE)
```

By names:

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

As name-value pairs:

```
y <- future({ slow_sum(x) }, globals =
                                list(slow_sum = slow_sum, x = rnorm(n = 100)))
```

Disable:

```
y <- future({ slow_sum(x) }, globals = FALSE)
```

A1.5 Full control of globals (implicit API)

Automatic (default):

```
x <- rnorm(n = 100)
y %<-% { slow_sum(x) } %globals% TRUE
```

By names:

```
y %<-% { slow_sum(x) } %globals% c("slow_sum", "x")
```

As name-value pairs:

```
y %<-% { slow_sum(x) } %globals% list(slow_sum = slow_sum, x = rnorm(n = 100))
```

Disable:

```
y %<-% { slow_sum(x) } %globals% FALSE
```

A1.6 Protection: Exporting too large objects

```
x <- lapply(1:100, FUN = function(i) rnorm(1024 ^ 2))
y <- list()
for (i in seq_along(x)) {
  y[[i]] <- future( mean(x[[i]]) )
}
```

gives error: "The total size of the 2 globals that need to be exported for the future expression ('mean(x[[i]])') is **800.00 MiB. This exceeds the maximum allowed size of 500.00 MiB (option 'future.globals.maxSize')**. There are two globals: 'x' (800.00 MiB of class 'list') and 'i' (48 bytes of class 'numeric')."

```
for (i in seq_along(x)) {
  x_i <- x[[i]]  ## Fix: subset before creating future
  y[[i]] <- future( mean(x_i) )
}
```

Comment: Interesting research project to automate via code inspection.

A1.7 Free futures are resolved

Implicit futures are always resolved:

```
a %<-% sum(1:10)
b %<-% { 2 * a }
print(b)
## [1] 110
```

Explicit futures require care by developer:

```
fa <- future( sum(1:10) )
a <- value(fa)
fb <- future( 2 * a )
```

For the lazy developer - not recommended (may be expensive):

```
options(future.globals.resolve = TRUE)
fa <- future( sum(1:10) )
fb <- future( 2 * value(fa) )
```

A1.8 What's under the hood?

- **Future class** and corresponding methods:
  - abstract S3 class with common parts implemented,
    e.g. globals and protection
  - new backends extend this class and implement core methods,
    e.g. **`value()`** and **`resolved()`**
  - built-in classes implement backends on top the parallel package

A1.9 Universal union of parallel frameworks

	future	parallel	foreach	batchtools	BiocParallel
	future	**parallel**	**foreach**	**batchtools**	**BiocParallel**
Synchronous	✓	✓	✓	✓	✓
Asynchronous	✓	✓	✓	✓	✓
Uniform API	✓		✓	✓	✓
Extendable API	✓		✓	✓	✓
Globals	✓		(✓)+**(soon by future)**		
Packages	✓				
Map-reduce ("lapply")	✓	✓	`foreach()`	✓	✓
Load balancing	✓	✓	✓	✓	✓
For loops	✓				
While loops	✓				
Nested config	✓				
Recursive protection	✓	mc	mc	mc	mc
RNG stream	✓+	✓	doRNG	(planned)	SNOW
Early stopping	✓				✓
Traceback	✓				✓

A2 Bells & whistles

A2.1 availableCores() & availableWorkers()

- **`availableCores()`** is a "nicer" version of **`parallel::detectCores()`** that returns the number of cores allotted to the process by acknowledging known settings, e.g.
  - **`getOption("mc.cores")`**
  - HPC environment variables, e.g. **`NSLOTS`**, **`PBS_NUM_PPN`**, **`SLURM_CPUS_PER_TASK`**, ...
  - **`_R_CHECK_LIMIT_CORES_`**
- **`availableWorkers()`** returns a vector of hostnames based on:
  - HPC environment information, e.g. **`PE_HOSTFILE`**, **`PBS_NODEFILE`**, ...
  - Fallback to **`rep("localhost", availableCores())`**

Provide safe defaults to for instance

```
plan(multiprocess)
plan(cluster)
```

A2.2: makeClusterPSOCK()

`future::makeClusterPSOCK()`:

- Improves upon **`parallel::makePSOCKcluster()`**
- Simplifies cluster setup, especially remote ones
- Avoids common issues when workers connect back to master:
  - uses SSH reverse tunneling
  - no need for port-forwarding / firewall configuration
  - no need for DNS lookup
- Makes option **`-l <user>`** optional (such that **`~/.ssh/config`** is respected)
- Automatically stop clusters when no longer needed, e.g. by garbage collector
- Automatically stop workers if set up of cluster fails; **`parallel::makePSOCKcluster()`** may leave background R processes behind

A2.3 HPC resource parameters

With 'future.batchtools' one can also specify computational resources, e.g. cores per node and memory needs.

```
plan(batchtools_sge, resources = list(mem = "128gb"))
y %<-% { large_memory_method(x) }
```

Specific to scheduler: `resources` is passed to the job-script template where the parameters are interpreted and passed to the scheduler.

Each future needs one node with 24 cores and 128 GiB of RAM:

```
resources = list(l = "nodes=1:ppn=24", mem = "128gb")
```

A2.4: Example: An academic cluster

One worker per compute node (6 workers total)

```
nodes <- c("cauchy", "leibniz", "bolzano",
           "shannon", "euler", "hamming")
plan(cluster, workers = nodes)

## Find our 80 FASTQ files
fastq <- dir(pattern = "[.]fq$")         ## 200 GB each

## Align the to human genome
bam <- listenv()
for (i in seq_along(fastq)) {
  bam[[i]] %<-% DNAseq::align(fastq[i])  ## 3 hours each
}
```

- **Total processing time: ~1.7 days = 40 hours**

A2.5 Example: An academic cluster

Four workers per compute node (24 workers total)

```
nodes <- c("cauchy", "leibniz", "bolzano",
           "shannon", "euler", "hamming")
plan(cluster, workers = rep(nodes, each = 4))

## Find our 80 FASTQ files
fastq <- dir(pattern = "[.]fq$")         ## 200 GB each

## Align the to human genome
bam <- listenv()
for (i in seq_along(fastq)) {
  bam[[i]] %<-% DNAseq::align(fastq[i])  ## 3 hours each
}
```

- **Total processing time: ~0.4 days = 10 hours** (cf. 40 hours and 10 days)

A2.6: Nested futures

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

- `plan(list(tweak(cluster, workers = nodes), multiprocess))`

```
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)
  }
}
```

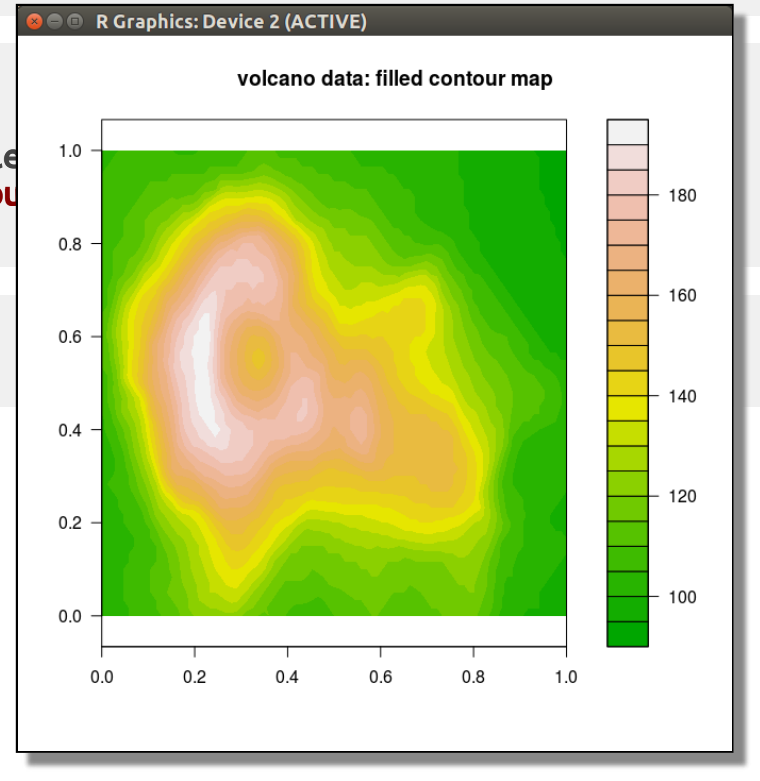
A3. More Examples

A3.1 Plot remotely - display locally

```
> library(future)
> plan(cluster, workers = "remote.org")
```

```
## Plot remotely
> g %<-% R.devices::capturePlot({
    filled.contour(volcano, color.pale
    title("volcano data: filled contou
 })
```

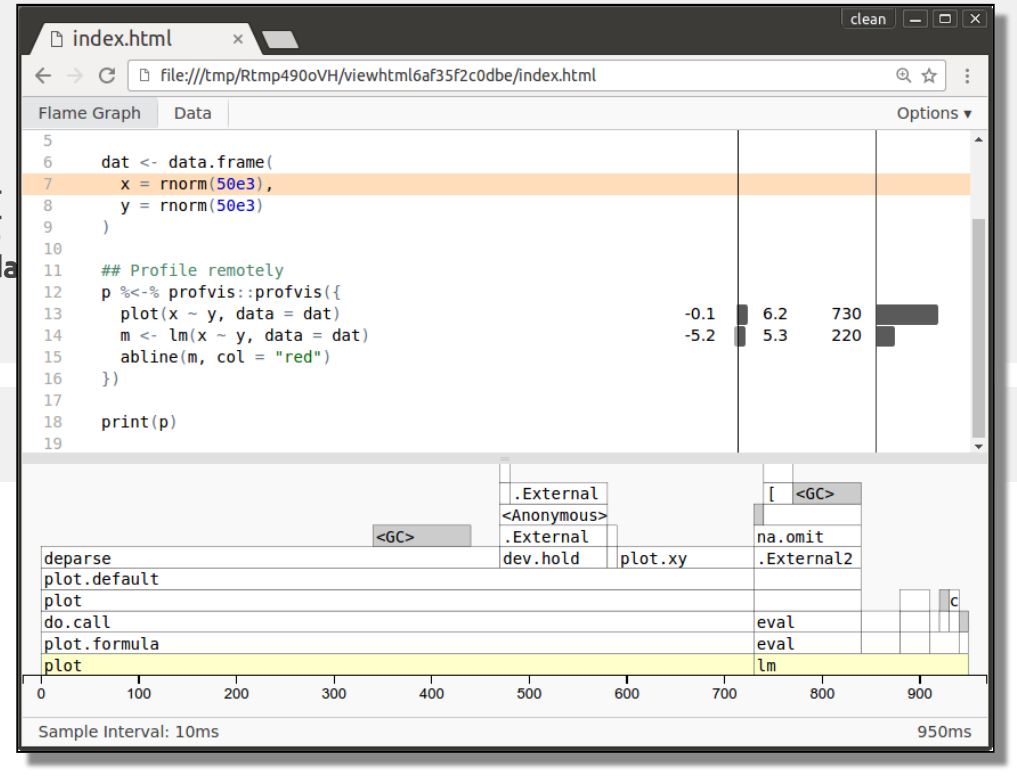
```
## Display locally
> g
```



A3.2 Profile code remotely - display locally

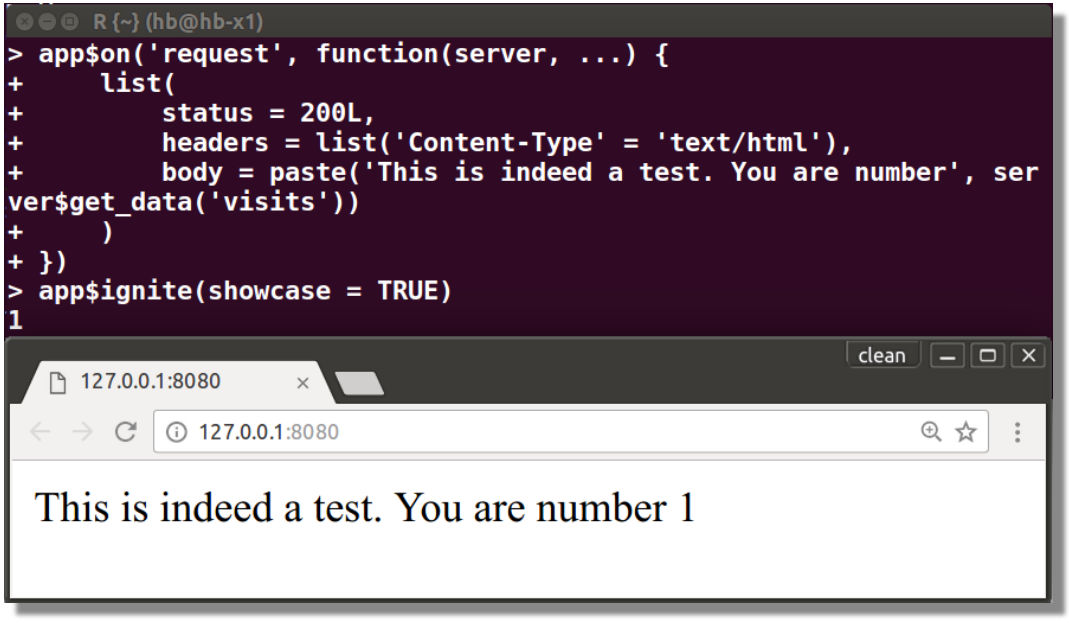
```
> plan(cluster, workers = "remote.org")

> dat <- data.frame(
+   x = rnorm(50e3),
+   y = rnorm(50e3)
+ )

## Profile remotely
> p %<-% profvis::profvis({
+   plot(x ~ y, data = dat)
+   m <- lm(x ~ y, data = da
+   abline(m, col = "red")
+ })
```

```
## Browse locally
> p
```



A3.3 *fiery* - flexible lightweight web server (Thomas Lin Pedersen)

"... **framework for building web servers in R. ... from serving static content to full-blown dynamic web-apps**"



A3.4 "It kinda just works" (furrr = future + purrr)

```
plan(multisession)
mtcars %>%
  split(.$cyl) %>%
  map(~ future(lm(mpg ~ wt, data = .x))) %>% values %>%
  map(summary) %>%
  map_dbl("r.squared")
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Comment: This approach not do load balancing. I have a few ideas how support for this may be implemented in future framework, which would be beneficial here and elsewhere.

A4. Future Work

A4.1 Standard resource types(?)

For any type of futures, the develop may wish to control:

- memory requirements, e.g. `future(..., memory = 8e9)`
- local machine only, e.g. `remote = FALSE`
- dependencies, e.g. `dependencies = c("R (>= 3.5.0)", "rio"))`
- file-system availability, e.g. `mounts = "/share/lab/files"`
- data locality, e.g. `vars = c("gene_db", "mtcars")`
- containers, e.g. `container = "docker://rocker/r-base"`
- generic resources, e.g. `tokens = c("a", "b")`
- ...?

Risk for bloating the Future API: Which need to be included? Don't want to reinvent the HPC scheduler and Spark.