Futures in R: Atomic Building Blocks for Asynchronous Evaluation

Henrik Bengtsson (Assoc Prof, MSc CS, PhD Math Stat)
Epidemiology & Biostatistics, Univ of California, San Francisco

CRAN packages: future, future.BatchJobs, doFuture, ...

Outline

1. Overview of futures and their implementation in R
2. Why a Future API?
3. Some features
4. Adding new backends
5. Richer parallel constructs on top of futures
6. What's under the hood?
7. Comparison to other parallel frameworks in R
8. Other usages of futures
9. Future work

Why do we parallelize software?

Parallel and distributed processing can be used to:

1. **speed up processing** (wall time)
   - multiple cores
   - multiple machines
2. **decrease memory footprint** (per machine)
   - multiple machines
3. **avoid data transfers**
   - compute where the data live
4. ...

(This talk focuses on the first two - will briefly touch on the third at the end.)

Definition: Future

- 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 either **unresolved** or **resolved** (= evaluated)

Standard R:

```
v <- expr
```

Explicit Future API:

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

Implicit Future API:

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

Friedman & Wise (1976, 1977), Hibbard (1976), Baker & Hewitt (1977)

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

Explicit API:

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

## Non-blocking setup of futures
> fa <- future( slow_sum(1:50)   )
> fb <- future( slow_sum(51:100) )
> 1:3
[1] 1 2 3

## Blocks until futures are resolved
> y <- value(fa) + value(fb)
> y
[1] 5050
```

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

Implicit API:

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

## Non-blocking setup of futures
> a %<-% slow_sum(1:50)
> b %<-% slow_sum(51:100)
> 1:3
[1] 1 2 3

## Blocks until futures are resolved
> y <- a + b
> y
[1] 5050
```

Many ways to resolve futures

Strategy:	
`sequential`	sequentially (default)
`multicore`	parallel on same machine (forks)
`multisession`	parallel on same machine (PSOCK)
`multiprocess`	either `multicore` or `multisession`
`cluster`	parallel across set of machines
...	...

```
> plan(multiprocess, workers = 2)
> plan(cluster, workers = c("n1", "n2.remote.org"))
```

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

R package: future

- "Write once, run anywhere"
- A simple **unified API** ("interface of interfaces")
- Works the same on **all platforms** (Unix, macOS, Windows)
- **Easy to install**
- **Lightweight** (~350 kB incl. dependencies; 1/3 is `digest` for md5)
- **Extendable** by anyone

```
+-------------------------------------------------------+
|                     < Future API >                    |
|-------------------------------------------------------|
|                          future                       |
|-------------------------------------------------------|
|            parallel             |  globals  | (listenv) |
|---------------------------------+-----------+-----------+
|   snow   |   Rmpi   |   nws     | codetools |
+---------------------------------+-----------+
```

Why a Future API?

Problem: heterogeneity

- R has various APIs for concurrent and parallel processing
- By choosing which to use, the developer limits users' options
- Introduction of new computational backends adds new APIs

```
y <- lapply(x, FUN = slow_sum)
```

```
y <- parallel::mclapply(x, FUN = slow_sum)
```

```
library("parallel")
cluster <- makeCluster(4)
y <- parLapply(cluster, x, fun = slow_sum)
stopCluster(cluster)
```

Why a Future API?

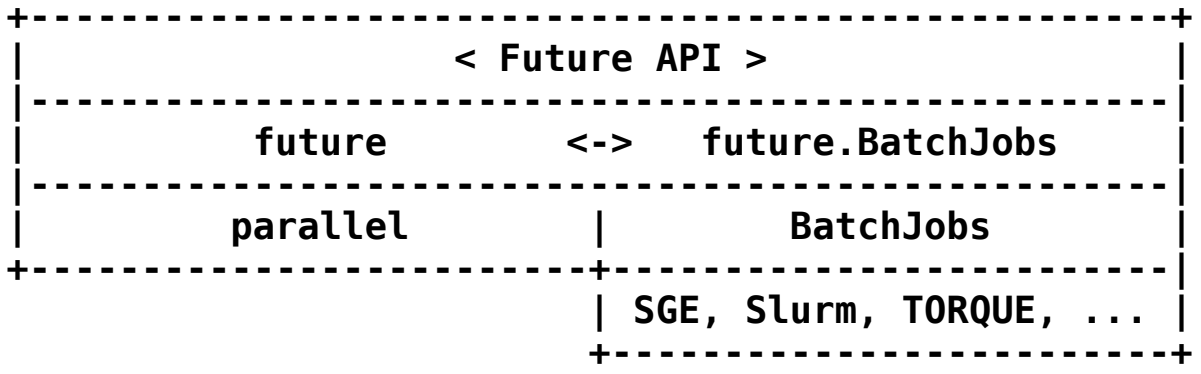
Solution: "interface of interfaces"

- The Future API encapsulates above heterogeneity:
  - fever decisions to be made by the developer
  - more options for the end user, i.e. where and how to run
- Developer decides *what to* parallelize - user decides *how to*
- Provides atomic building blocks for richer parallel constructs, e.g. map-reduce evaluation and foreach.
- Can be extended for new computational backends, e.g. future.BatchJobs.

Comment: The wide adoption of the foreach API shows there is great demand for this type of "interface of interfaces".

R package: future.BatchJobs

- BatchJobs ("now" batchtools) provides a map-reduce API for HPC schedulers, e.g. LSF, OpenLava, SGE, Slurm, and TORQUE / PBS
- future.BatchJobs implements the **Future API** on top of BatchJobs




```
> library("future.BatchJobs")
> plan(batchjobs_sge)

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

Real-world example

DNA-sequence data files from 100's of individual each being a few hundred GiB:s large. On a single-core machine, total processing time may be many hours per individual.

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

bam <- listenv()
for (i in seq_along(fastq)) {
  bam[[i]] %<-% DNAseq::align(fastq[i])
}
bam <- as.list(bam)
```

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

Comment: The use of `listenv` is non-critical and only needed for implicit futures when assigning them by index (instead of by name).

Nested futures

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

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

- `plan(batchjobs_sge)`
- `plan(list(batchjobs_sge, sequential))`
- `plan(list(batchjobs_sge, multiprocess))`

HPC resource parameters

With future.(BatchJobs|batchtools) one can also specify computational resources, e.g. cores per node and memory needs.

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

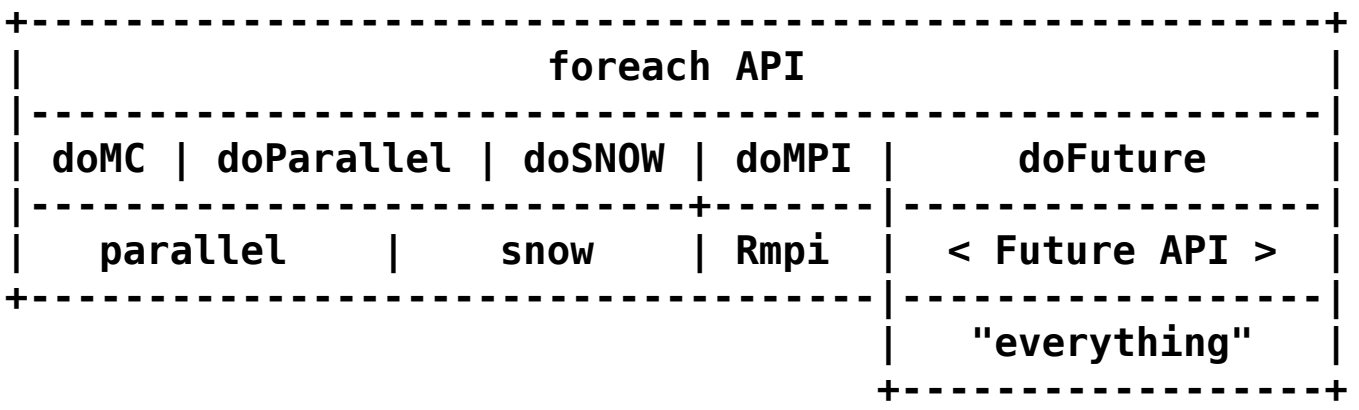
Specific to scheduler: `resources` is passed to the job-script template where the parameters are interpreted and passed to the scheduler.

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

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

R package: doFuture

- A **foreach** adaptor on top of the Future API
- Allows foreach to utilize **all future-compatible backends**
- Specifically, **HPC schedulers** can now be used

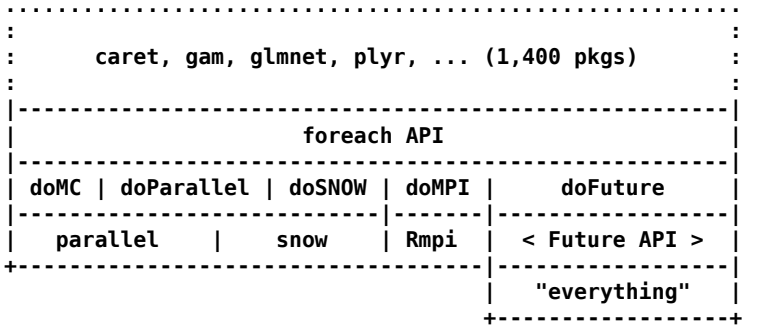



```
library("doFuture")
registerDoFuture()
plan(batchjobs_sge)

y <- foreach(i = 1:3) %dopar% { ... }
```

1,400+ packages can now parallelize on HPC

350 CRAN & Bioc packages that depend directly on foreach, and another 1,150 indirectly, can **now utilize HPC clusters**.




```
library("doFuture")
registerDoFuture()      ## (a) Tell foreach to use futures

library("future.BatchJobs")
plan(batchjobs_slurm)   ## (b) Resolve via Slurm scheduler
```

```
library("plyr")                    ## Uses foreach internally
fastq <- dir(pattern = "[.]fq$")
bam <- llply(fastq, DNAseq::align, .parallel = TRUE)
```

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
- **Global variables**
- **Protection** (against user "mistakes")
  - too large global exports
  - recursive parallelism

Future takes care of globals

- **Global (aka "free") variables and functions** needed for the future expression to be resolved are by default **identified automatically** and **exported**
- Dependent **packages** are automatically loaded
- **Static-code inspection** by walking the AST (recursively)

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

Globals identified and exported (or frozen):

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

Full control of globals

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)
```

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
```

False-negative and 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.

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.

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) )
```

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
```

Comment: In future (<= 1.2.0), it was possible for end users to control eager versus lazy evaluation via `plan()`, but that made it very hard for the developer.

Future: Universal union of parallel frameworks

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

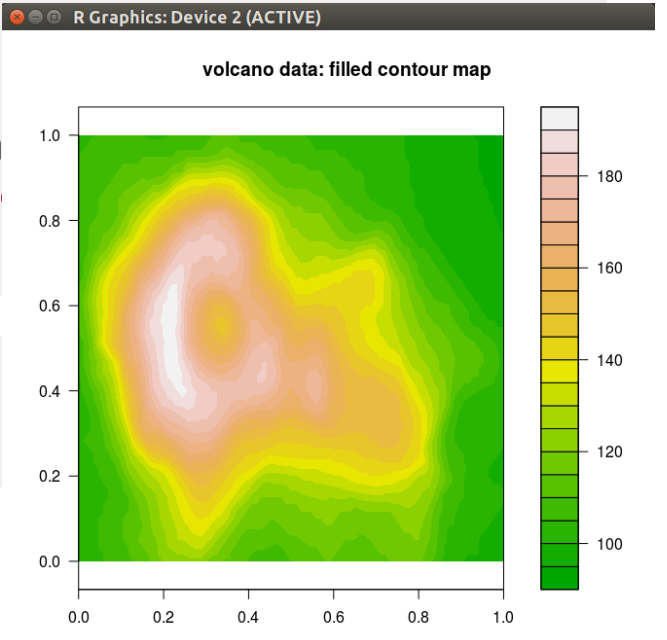
Other usages of future

Plot remotely - display locally

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

```
## Plot remotely
> g %<-% R.devices::capturePlot({
+   filled.contour(volcano, color.pa
+   title(main = "volcano data: fill
+ })
```

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



modulr - "make for R"

```
library("modulr")  ## https://github.com/aclemen1/modulr

"foo" %provides% { "Hello" }

"bar" %provides% { "World" }

"foobar" %requires% list(f = "foo", b = "bar") %provides% {
  paste0(f, " ", b, "!")
}
```

```
future::plan("multiprocess")
make("foobar")
# [09:29:07] Making 'foobar' ...
# [09:29:07] * Visiting and defining dependencies ...
# [09:29:07] * Constructing dependency graph ... OK
# [09:29:07] * Sorting 2 dependencies with 2 relations ...
# [09:29:07] * Evaluating new and outdated dependencies ...
# [09:29:07] ** [... parallel processing ...]
# [09:29:07] DONE ('foobar')
[1] "Hello World!"
```

Future work

Global variables

- Memoization of which globals exist

Futures

- Terminating futures (local and remote signalling)
- Progress updates, e.g. progress bars
- Capturing stdout and stderr uniformly
- On the-fly time and memory benchmark statistics

Future work

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 (>= 3.4.0)", "rio"))`
- file-system availability, e.g. `mounts = "/share/lab/files"`
- data locality, e.g. `vars = c("gene_db", "mtcars")`
- containers, e.g. `container = "singularity:hb/r-base"`
- generic resources, e.g. `tokens = c("a", "b")`
- ...?

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

Futures I'd like to see

- `library(future.batchtools)` - soon on CRAN!
- `plan(aws_lambda)`
  - Short high-burst compute on Amazon "serverless" AWS Lambda
  - Baby steps taken with Shaun Jackman (U of British Colombia)
- `plan(rcpp)`
  - Identify a subset of the R language that can be transpiled to Rcpp
  - On-the-fly transpile-and-compile an R expression into Rcpp
  - Ex: `sum %<-% ({ y <- 0; for (i in 1:1e6) y <- y + x[i] })`
- `plan(p2p)`
  - Private and / or community-based peer-to-peer computer cluster
  - "Sandboxed" exec of R in Linux containers on friends' computers

Summary of features and objectives

- Unified API (synchronous and asynchronous)
- For beginners as well as advanced users
- Portable code (invariant to backend)
- Developer decides what to parallelize - user decides how to
- Nested parallelism on nested heterogeneous backends
- Protect against infinite, recursive parallelism
- Unified handling of globals (invariant to backend)
- Protect against trying to export too large globals
- Friendly defaults (e.g. Appendix A1)
- Friendly to multi-tenant HPC clusters (e.g. Appendix A2)

Building a better future

I  feedback, bug reports, concerns, and suggestions

https://github.com/HenrikBengtsson/future

@HenrikBengtsson

Thank you!

Appendix

A1. Bells & whistles: makeClusterPSOCK()

`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 (so `~/.ssh/config` is respected)

A2. 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`, ...
  - `_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 e.g.

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

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

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

With futures:

```
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
```

A4. Google Cloud Engine Cluster

```
## Launch ten R Docker containers on GCE
vms <- lapply(paste0("node", 1:10),
              FUN = googleComputeEngineR::gce_vm,
              template = "r-base")
cl <- as.cluster(lapply(vms, FUN = gce_ssh_setup),
            docker_image = "henrikbengtsson/r-base-future")
```

```
library("future")
plan(cluster, workers = cl)  ## Resolve all futures on GCE

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

print(pi_hat)
## 3.1415
```

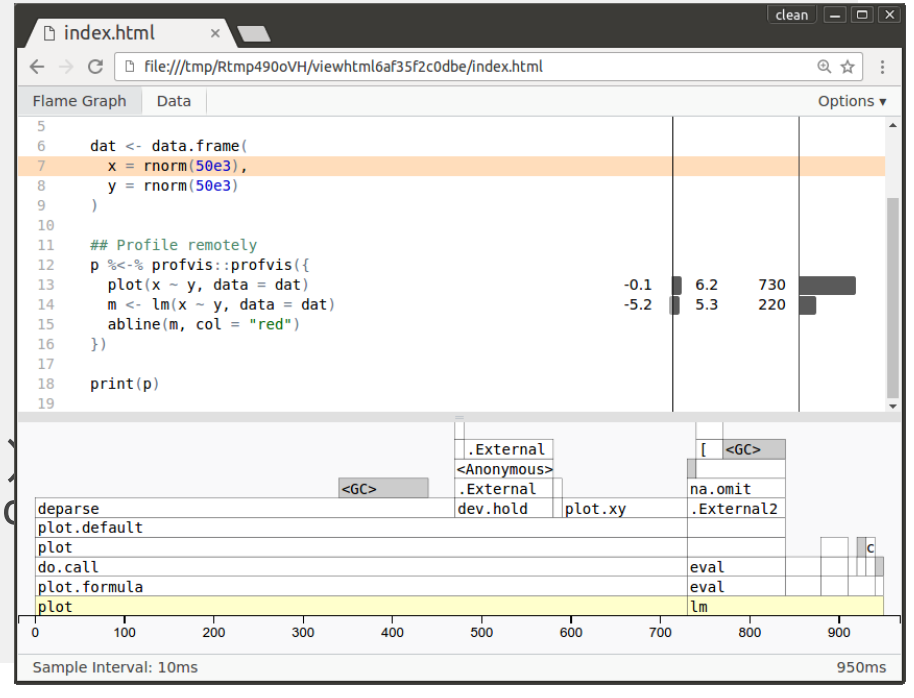
A5. Profile code remotely - display locally

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

> library("profvis")

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

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




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