

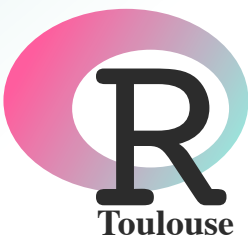
# Tools for parallelizing R code easily

R-Toulouse

---

**Boris Hejblum**

**April 9<sup>th</sup>, 2026**



# First things first: required packages

---

Install the missing packages

```
req_pkgs ← c("future.apply", "pbapply", "mirai", "microbenchmark")
install.packages(req_pkgs[!req_pkgs %in% installed.packages()[, "Package"]])
```

Load them

```
library(future)
library(future.apply)
library(pbapply)
library(mirai)
library(microbenchmark)
```

# Introduction to parallel execution

---

Parallel code execution enables **faster computation** by leveraging multiple CPU cores.

The *total* amount of work is not reduced — but it gets done **quicker**.

Many algorithms are "*embarrassingly parallel*": they can be decomposed into completely independent calculations.

In Statistics, it is natural to *parallelize over observations and/or parameters*.

Typically, **for**-loops are the most common and straightforward candidates for parallelization.

# Parallel computation — steps

---

## Operations required to run a parallel computation:

1. Start  $m$  *worker* processes (CPUs)
2. Send the necessary data and functions to the *workers*
3. Split the task into  $m$  sub-tasks of similar magnitude
4. Wait for all *workers* to finish
5. Gather and combine the results from the different *workers*
6. Stop the *worker* processes

# Communication protocols

Several protocols for communicating between *workers* are available, depending on the OS:

Protocol	Platform	Notes
Fork	Unix (Mac & Linux)	Fast — shares memory with parent process
PSOCK	All (incl. Windows)	Spawns fresh R sessions; more overhead
NNG sockets	All	C-level; ultra-low latency (used by <code>mirai</code> )
MPI	HPC clusters	For distributed multi-node computing

`pbapply`, `future` and `mirai` expose a unified syntax regardless of the underlying backend.

```
future.apply
```

---

## Parallel apply

# First parallel function

## 👉 Your turn!

1. Check available cores: `future::availableCores()`
2. Declare a parallel plan — leave **at least one core free**:  
`future::plan(multisession, workers = XX)`
3. Use `future.apply::future_sapply()` to compute `log()` of  $n$  numbers in parallel
4. Wrap into a function `mylog_parallel()`
5. Compare execution times on `1:100`:  
`microbenchmark(mylog_parallel(1:100), mylog(1:100), log(1:100), times = 10)`

## First parallel function — solution

```
future::availableCores()-1 #nb available cpus

mylog_parallel ← function(x) {
  future_sapply(1:length(x),
               FUN = function(i){
                 log(x[i])
               }
  )
}

plan(multisession, workers = 2)
mylog_parallel(1:100)
mb ← microbenchmark(mylog_parallel(1:100), mylog(1:100), log(1:100), times = 10)
plan(sequential)
```


```
#> Unit: nanoseconds
#>          expr      min      lq    mean  median     uq    max neval cld
#> mylog_parallel(1:100) 8094261 8397702 8916946.5 8568816 9479241 10637409   10  a
#>      mylog(1:100)    4018    4879 150228.1   5125    6560 1453860    10  b
#>      log(1:100)     328     697  1102.9   1107    1517   1722    10  b
```

# Parallelization is not always effective

---

## The parallel code is much slower..

Because each individual computation is too quick,  spends more time on **inter-process communication** than on actual computation.

 **Rule of thumb:** one iteration of the loop must be long enough for parallelization to pay off.  
Communication overhead is roughly constant – it only becomes negligible when the task itself takes time.

## A realistic example

---

**Bootstrap confidence intervals** for the mean of each of **500 variables** (e.g. omics measurements), using **B = 2 000 bootstraps**

A single bootstrap is fast, but repeating it  $100 \times 2\,000$  times makes this *embarrassingly parallel, and slow enough for parallelism to pay off*.

```
set.seed(942026)
n_obs  <- 50      # observations per variable
n_vars <- 200     # number of variables
B      <- 2000    # bootstrap resamples

mydata <- matrix(rnorm(n_obs * n_vars), nrow = n_obs, ncol = n_vars)
```

Computing a **bootstrap CI** at 5% for one variable:

1. resample B times
2. compute the mean each time
3. finally take the 2.5 % and 97.5 % quantiles of the overall results.

```
boot_ci <- function(col, B = 2000) {
  means <- replicate(B, mean(sample(col, replace = TRUE)))
  quantile(means, c(0.025, 0.975))
}
```

## apply and for-loops

Experienced R users often reach for `*apply()` functions:

```
bootci_apply ← function(x, B = 2000){  
  apply(X = x, MARGIN = 2, FUN = boot_ci, B = B)  
}  
system.time(bootci_apply(mydata))
```

```
#>   user  system elapsed  
#> 4.893   0.077   4.972
```

Yet, a well-written `for`-loop is *equally fast*:

```
bootci_for ← function(x, B = 2000) {  
  ans ← matrix(NA_real_, nrow = 2, ncol = ncol(x))  
  for (i in seq_len(ncol(x))) ans[, i] ← boot_ci(x[, i], B)  
  ans  
}  
system.time(bootci_for(mydata))
```

```
#>   user  system elapsed  
#> 4.898   0.071   4.971
```

## Efficient parallelization — `future.apply`

### 👉 Your turn!

Parallelize the bootstrap CI computation using `future_sapply()`.

Does it improve computation time compared to the sequential versions this time ?

```
bootci_par ← function(x, B = 2000) {  
  future_sapply(seq_len(ncol(x)), FUN = function(i) boot_ci(x[, i], B),  
               future.seed=TRUE)  
}  
  
n_workers ← 3 #future::availableCores() - 1  
plan(multisession, workers = n_workers)  
system.time(bootci_par(mydata))  
plan(sequential)
```

```
pbapply
```

---

**Parallel apply with a progress bar**

## pbapply — progress bar + parallelizm

`pbapply` is a thin wrapper that adds a **progress bar** to any `*apply` call.

Its `cl` argument accepts either:

- an integer (number of cores) → creates a PSOCK cluster internally, or
- an existing `cluster` object from `parallel::makeCluster()`.

```
library(pbapply)
?pbapply # same signature as lapply + cl =
```

### 👉 Your turn!

Rewrite `bootci_apply` using `pbapply::pbsapply()` with `cl = future::availableCores() - 1`.

Add it to the benchmark. Does it behave differently from `future.apply`?

## pbapply — solution

```
bootci_pb ← function(x, cl, B = 2000) {  
  pbapply::pbsapply(1:ncol(x), function(i){  
    boot_ci(x[, i], B)  
  },  
  cl = cl)  
}  
  
system.time(bootci_pb(mydata, cl = 2))
```

```
#>   user  system elapsed  
#> 0.001  0.015  1.088
```

```
pboptions(nout=25) #default: nout=100  
pboptions(type="timer") #default: type="timer"  
system.time(bootci_pb(mydata, cl = n_workers))
```

```
#>   user  system elapsed  
#> 0.638  0.070  0.716
```

The progress bar induces some overhead, that can get noticed for fast subtasks

`mirai`

---

**Minimal, async, lightning-fast**

# mirai — why another package?

mirai (Japanese for "future") takes a different architectural approach:

	future	mirai
<b>Backend</b>	R sessions (PSOCK/Fork)	NNG C sockets
<b>Overhead</b>	Moderate	Minimal
<b>Style</b>	Synchronous ( <code>plan</code> + <code>*apply</code> )	Async-first ( <code>mirai()</code> returns immediately)
<b>Best for</b>	General parallel work	High-frequency tasks, Shiny, production
<b>OS</b>	All	All

💡 mirai tasks are **non-blocking**: your R session remains free while workers compute. Results are fetched explicitly when you need them.

## mirai — core API

---

**Set up workers** (analogous to `plan(multisession(workers = n))`):

```
library(mirai)
daemons(n = 4)      # start 4 persistent workers
```

**Launch an async task** — pass extra data as named arguments:

```
m ← mirai({ median(col) }, col = mydata[, 1])
# returns immediately — R is not blocked!
```

**Retrieve the result** — blocks only at this point:

```
m[]      # waits and returns the value
```

**Shut down workers:**

```
daemons(0)
```

## mirai — your turn!

### 👉 Your turn!

1. Install and load `mirai`
2. Start daemons: `daemons(future::availableCores() - 1)`
3. Write `colmedian_mirai()` using `mirai_map()`
4. Add it to the benchmark — compare with `future.apply` and `pbapply`
5. Shut down: `daemons(0)`

## mirai — solution

```
daemons(n_workers)

bootci_mirai ← function(x, B = 2000) {
  ms ← lapply(seq_len(ncol(x)), function(i) {
    mirai({
      means ← replicate(B, mean(sample(col, replace = TRUE)))
      quantile(means, c(0.025, 0.975))
    }, col = x[, i], B = B)
  })
  sapply(ms, function(m) m[])
}

system.time(bootci_mirai(mydata))
```

```
#>   user  system elapsed
#> 0.005   0.012   0.719
```

```
daemons(0) # clean up workers
```

# mirai — full benchmark

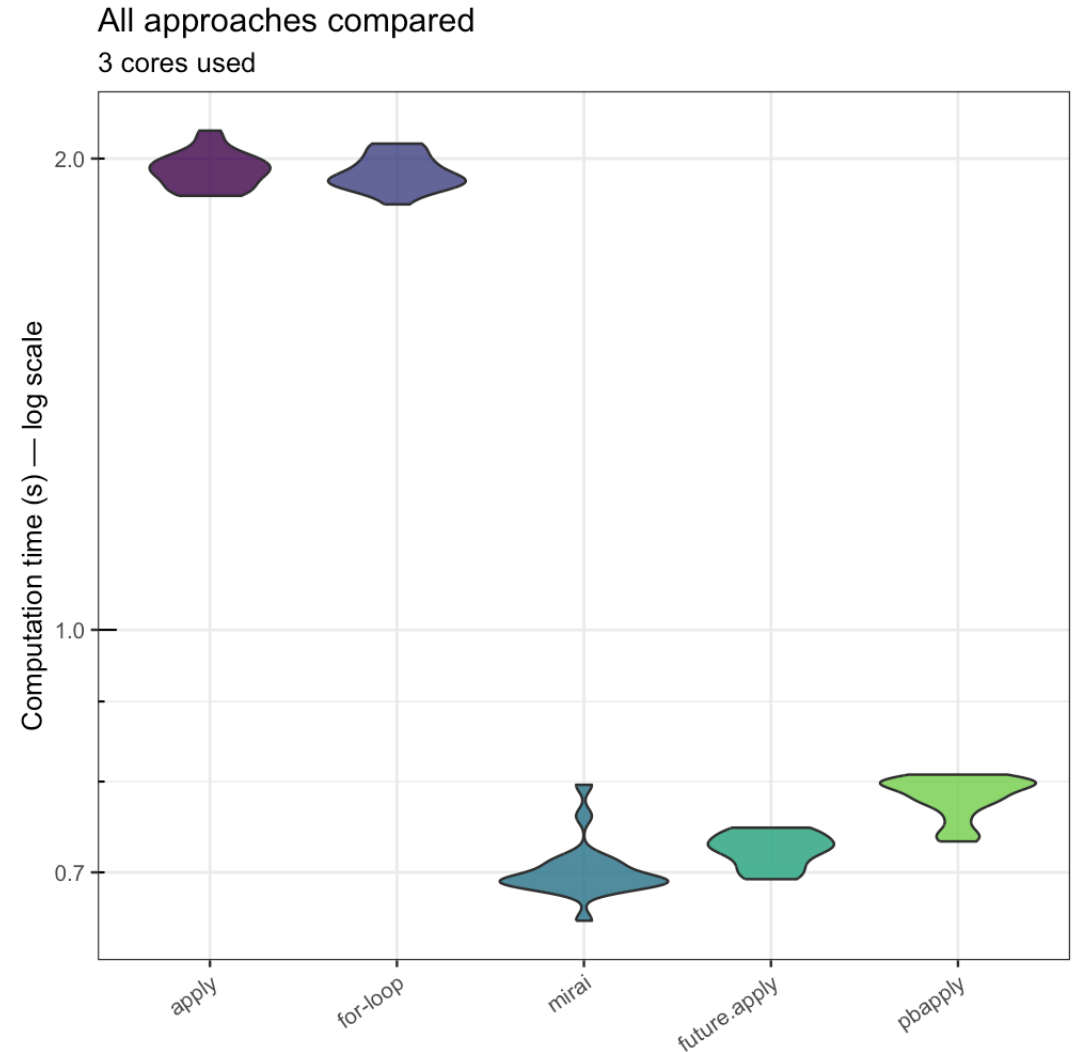
```

daemons(n_workers)
plan(multisession, workers = n_workers)
bootci_mirai(mydata) # warm-up call
mb3_a ← microbenchmark(
  "apply" = bootci_apply(mydata),
  "for-loop" = bootci_for(mydata),
  "mirai" = bootci_mirai(mydata),
  times = 20
)
daemons(0) #cleanup

plan(multisession, workers = n_workers)
bootci_par(mydata) # warm-up call
bootci_pb(mydata, cl = n_workers) # warm-up call
mb3_b ← microbenchmark(
  "future.apply" = bootci_par(mydata),
  "pbapply" = bootci_pb(mydata, cl = n_workers),
  times = 20
)
plan(sequential) #cleanup

mb3 ← rbind.data.frame(mb3_a, mb3_b)

```



## Summary & conclusion

Situation	Recommended
Quick script, want a progress bar	<code>pbapply</code>
General parallel work, drop-in replacement	<code>future</code> or <code>future.apply</code>
High-throughput / async / Shiny integration	<code>mirai</code>

parallelization **can** speed up computation – but the gain depends on the ratio between:

- **communication time** (fixed cost) *VS*
- **computation time per task** (must dominate)

## Going Further

---

More details on R packages and advanced R programming:

👉 Online lecture notes from [endeav<sup>R</sup> course: a course on code efficiency and software development with R](#)

A list of additional, less straightforward (and not more efficient computationally), <sup>R</sup> packages for parallel computations:

- `doParallel`
- `foreach`
- `itertools`
- `batchtools`