Tools for parallelizing R code easily

R-Toulouse

Boris Hejblum

April 9th, 2026

Toulouse

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 `mirai`)
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

Toulouse

👉 **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
n_workers ← 3

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

plan(multisession, workers = n_workers)
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) 7845842 7921282 8515622.1 7985549.5 9247632 9923763    10  a
#>           mylog(1:100)    3731    3936  129035.2    4100.0    6765 1246523    10   b
#>             log(1:100)     492     615    1184.9    1168.5    1681    2214    10   b
```

Parallelization is not always effective

The parallel code is much slower...

Because each individual computation is too quick, R 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 **200 variables** (e.g. omics measurements), using **B = 2 000 bootstraps**

A single bootstrap is fast, but repeating it 200 × 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
#>   1.980   0.029   2.011
```

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 1:ncol(x)){
    ans[, i] ← boot_ci(x[, i], B)
  }
  return(ans)
}
system.time(bootci_for(mydata))
```

```
#>    user  system elapsed
#>   1.973   0.021   1.997
```

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_future ← function(x, B = 2000) {
  future_sapply(1:ncol(x), FUN = function(i) {
    boot_ci(x[, i], B)
  },
  future.seed = TRUE) #ensures parallel-safe RNG
}

n_workers ← 3 #future::availableCores() - 1
system.time(bootci_future(mydata))
plan(multisession, workers = n_workers)
system.time(bootci_future(mydata))
plan(sequential)
```

Toulouse

`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)
?pblapply   # 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 = n_workers))
```

```
#>    user  system elapsed
#>   0.656   0.085   0.755
```

```
pboptions(nout=25) #default: nout=100
pboptions(type="none") #default: type="timer"
system.time(bootci_pb(mydata, cl = n_workers))
```

```
#>    user  system elapsed
#>   0.672   0.094   0.769
```

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

Toulouse

`mirai`

Minimal, async, lightning-fast

`mirai` — why another package?

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

	`future`	`mirai`
Backend	R sessions (PSOCK/Fork)	NNG C sockets
Overhead	Moderate	Minimal
Style	Synchronous (`plan` + `*apply`)	Async-first (`mirai()` returns immediately)
Best for	General parallel work	High-frequency tasks, Shiny, production
OS	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 = 3)          # start 3 persistent workers
```

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

```
m ← sapply(1:ncol(mydata), function(i) mirai(boot_ci(mydata[, i]), .args=list(boot_ci=boot_ci, mydata=mydata, i=i)))
# returns immediately — R is not blocked!
```

Retrieve the result — blocks only at this point:

```
sapply(m, function(mm) mm[])          # waits and returns the value
```

Shut down workers:

```
daemons(0)
```

`mirai` — your turn!

👉 Your turn!

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

`mirai` — solution

```
bootci_mirai ← function(x, B = 2000) {
  ms ← lapply(1:ncol(x), function(i) {
    mirai(boot_ci(col = x[, i], B = B),
          .args=list(boot_ci=boot_ci, x=x, i=i, B=B))
  })
  sapply(X = ms, FUN = function(m) m[])
}

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

```
#>    user  system elapsed
#>   0.281   0.536   5.062
```

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

```
#>    user  system elapsed
#>   0.012   0.041   0.751
```

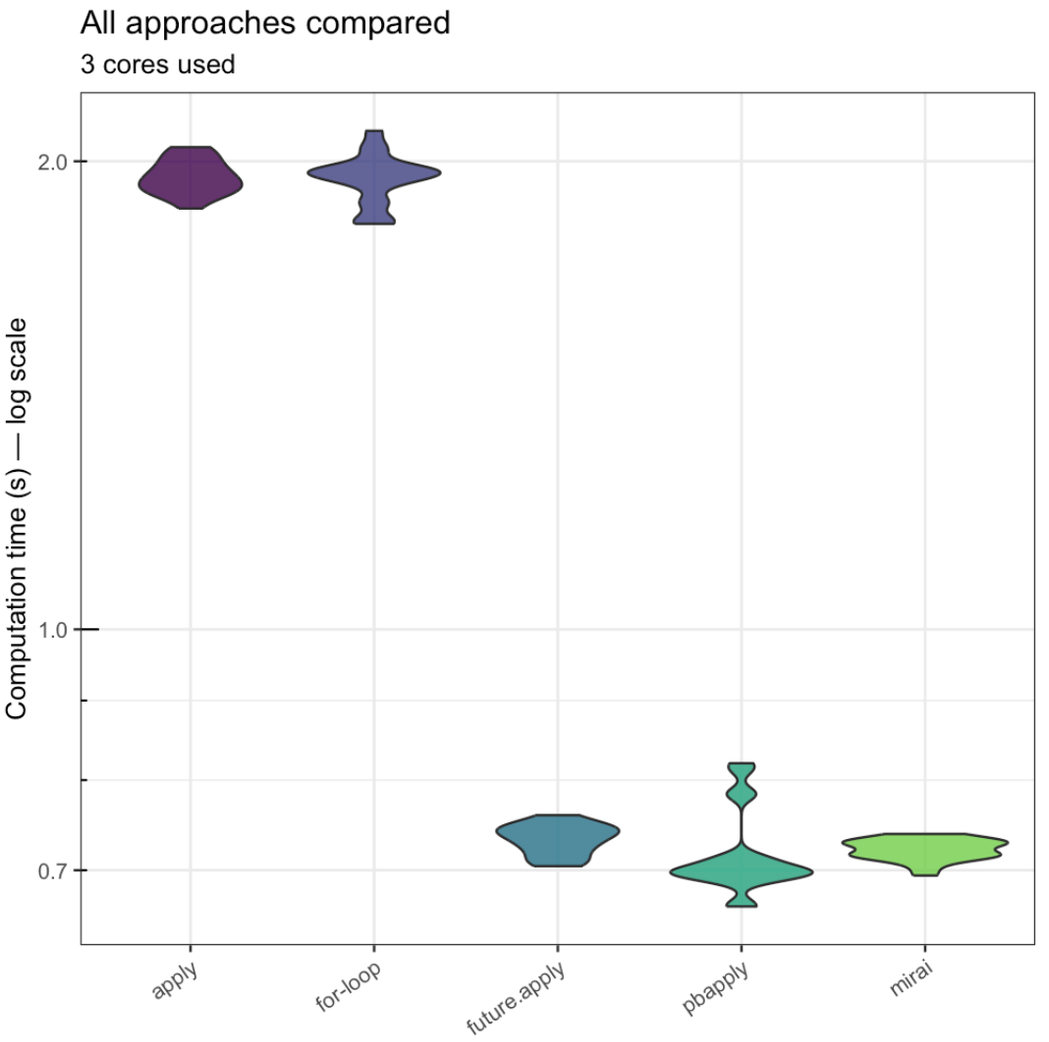
```
daemons(0)   # clean up workers
```

`mirai` — full benchmark

```
plan(multisession, workers = n_workers)
bootci_future(mydata) # warm-up call
bootci_pb(mydata, cl = n_workers)  # warm-up call
mb3_a ← microbenchmark(
  "apply" = bootci_apply(mydata),
  "for-loop" = bootci_for(mydata),
  "future.apply" = bootci_future(mydata),
  "pbapply"  = bootci_pb(mydata, cl = n_workers),
  times = 20
)
plan(sequential) #cleanup

daemons(n_workers)
bootci_mirai(mydata) # warm-up call
mb3_b ← microbenchmark(
  "mirai" = bootci_mirai(mydata),
  times = 20
)
daemons(0) #cleanup

mb3 ← rbind.data.frame(mb3_a, mb3_b)
```



Summary & conclusion

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

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 endeavR course: *a course on code efficiency and software development with R*

`futurize` provides an ever lighter API to leverage `future` parallelization.

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

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