Package 'future.mirai'

October 24, 2026

Version 1.1.0

Depends future (>= 1.49.0)

Imports mirai (>= 2.2.0), parallelly, utils

Suggests future.tests, future.apply, listenv

Title A 'Future' API for Parallel Processing using 'mirai'

Description Implementation of the 'Future' API <doi:10.32614/RJ-2021-048> on top of the 'mirai' package <doi:10.5281/zenodo.7912722>. By using this package, you get to take advantage of the benefits of 'mirai' plus everything else that 'future' and the 'Futureverse' adds on top of it. It allows you to process futures, as defined by the 'future' package, in parallel out of the box, on your local machine or across remote machines. Contrary to back-ends relying on the 'parallel' package (e.g. 'multisession') and socket connections, 'mirai_cluster' and 'mirai_multisession', provided here, can run more than 125 parallel R processes. As a reminder, regardless which future backend is used by the user, the code does not have to change, it gives identical results, and behaves exactly the same.

License Apache License (>= 2)

Language en-US

Encoding UTF-8

URL https://future.mirai.futureverse.org,
https://github.com/futureverse/future.mirai

BugReports https://github.com/futureverse/future.mirai/issues

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-7579-5165>),
Charlie Gao [ctb] (ORCID: <https://orcid.org/0000-0002-0750-061X>,
note: For 'mirai'-related patches and implementing feature requests
in 'mirai')

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2026-10-24 03:00:02 UTC

Contents

future.mirai 2
MiraiFutureBackend 3
mirai_cluster 3
mirai_multisession 6

Index **8**

`future.mirai` *future.mirai: A Future API for Parallel Processing using 'mirai'*

Description

The **future.mirai** package implements the Future API using the **mirai** package.

Author(s)

Maintainer: Henrik Bengtsson <henrikb@braju.com> (ORCID) [copyright holder]

Authors:

- Henrik Bengtsson <henrikb@braju.com> (ORCID) [copyright holder]

Other contributors:

- Charlie Gao <charlie.gao@shikokuchuo.net> (ORCID) (note: For 'mirai'-related patches and implementing feature requests in 'mirai') [contributor]

See Also

Useful links:

- https://future.mirai.futureverse.org
- https://github.com/futureverse/future.mirai
- Report bugs at https://github.com/futureverse/future.mirai/issues

Examples

```
library(future)
plan(future.mirai::mirai_multisession)

# A function that returns a future
# (note that N is a global variable)
f <- function() future({
  4 * sum((runif(N) ^ 2 + runif(N) ^ 2) < 1) / N
}, seed = TRUE)

# Run a simple sampling approximation of pi in parallel using  M * N points:
N <- 1e6  # samples per worker
M <- 10   # iterations
```

```
pi_est <- Reduce(sum, Map(value, replicate(M, f()))) / M
print(pi_est)

## Switch back to sequential processing, which also
## shuts down the automatically launched mirai workers
plan(sequential)
```

`MiraiFutureBackend` *A future backend based on the 'mirai' framework*

Description

Set up the future parameters.

Usage

```
MiraiFutureBackend(...)
```

Arguments

`...`	Additional arguments passed to `Future()`.

Value

An object of class `MiraiFutureBackend`.

`mirai_cluster` *Mirai-based cluster futures*

Description

WARNING: This function must never be called. It may only be used with `future::plan()`

Usage

```
mirai_cluster(..., envir = parent.frame())
```

Arguments

`envir`	The environment from where global objects should be identified.
`...`	Not used.

Value

Nothing.

Launch mirai workers via HPC job scheduler

If you have access to a high-performance-compute (HPC) environment with a job scheduler, you can use **future.mirai** and **mirai** to run parallel workers distributed on the computer cluster. How to set these workers up is explained in `mirai::cluster_config()`, which should work for our most common job schedulers, e.g. Slurm, Sun/Son of/Oracle/Univa/Altair Grid Engine (SGE), OpenLava, Load Sharing Facility (LSF), and TORQUE/PBS.

Note: Not all compute clusters support running **mirai** *workers this way. This is because* **mirai** *workers need to establish a TCP connection back to the machine that launched the workers, but some systems have security policies disallowing such connections from being established. This is often configured in the firewall and can only be controlled by the admins. If your system has such rules, you will find that the mirai jobs are launched and running on the scheduler, but we will wait forever for the mirai workers to connect back, i.e.* `mirai::info()[["connections"]]` *is always zero in the below example.*

Briefly, to launch a cluster of mirai workers on an HPC cluster, we need to:

1. configure `mirai::cluster_config()` for the job scheduler,
2. use configuration to launch workers using `mirai::daemons()`.

The first step is specific to each job scheduler and this is where you control things such as how much memory each worker gets, for how long they may run, and which environment modules to load, if any. The second step is the same regardless of job scheduler. Here is an example for how to run parallel mirai workers on a Slurm scheduler and then use these in futureverse.

```
# Here we give each worker 200 MiB of RAM and a maximum of one hour
# to run. Unless we specify '--cpus-per-task=N', each mirai worker
# is allotted one CPU core, which prevents nested parallelization.
# R is provided via environment module 'r' on this cluster.
config <- mirai::cluster_config(command = "sbatch", options = "
  #SBATCH --job-name=mirai
  #SBATCH --time=01:00:00
  #SBATCH --mem=200M
  module load r
")

# ------------------------------------------------------------------
# Launch eight mirai workers, via equally many jobs, wait for all of
# them to become available, and use them in futureverse
# ------------------------------------------------------------------
workers <- 8
mirai::daemons(n = workers, url = mirai::host_url(), remote = config)
while(mirai::info()[["connections"]] < workers) Sys.sleep(1.0)
plan(future.mirai::mirai_cluster)

# Verify that futures are resolved on a compute node
f <- future({
  data.frame(
     hostname = Sys.info()[["nodename"]],
           os = Sys.info()[["sysname"]],
```

```
    osVersion = utils::osVersion,
        cores = unname(parallelly::availableCores()),
      modules = Sys.getenv("LOADEDMODULES")
  )
})
info <- value(f)
print(info)
#>   hostname    os                osVersion cores  modules
#> 1      n12 Linux Linux Ubuntu 24.04.3 LTS     1  r/4.5.2

# Shut down parallel workers
plan(sequential)
mirai::daemons(0)
```

If you are on SGE, you can use the following configuration:

```
config <- mirai::cluster_config(command = "qsub", options = "
  #$ -N mirai
  #$ -j y
  #$ -cwd
  #$ -l h_rt=01:00:00
  #$ -l mem_free=200M
  module load r
")
```

Everything else is the same.

Comment: `mirai::cluster_config()` configures the jobs to run vanilla POSIX shells, i.e. `/bin/sh`. This might be too strict for some users. If your setup requires your jobs to be run using Bash (`/bin/bash`), you can tweak the configuration `config` object manually to do so;

```
config$command <- "/bin/bash"
config$args <- sub("/bin/sh", config$command, config$args)
```

Examples

```
library(future)

# Manually launch mirai workers
mirai::daemons(parallelly::availableCores())

plan(future.mirai::mirai_cluster)

# A function that returns a future
# (note that N is a global variable)
f <- function() future({
  4 * sum((runif(N) ^ 2 + runif(N) ^ 2) < 1) / N
}, seed = TRUE)

# Run a simple sampling approximation of pi in parallel using  M * N points:
```

```
N <- 1e6  # samples per worker
M <- 10   # iterations
pi_est <- Reduce(sum, Map(value, replicate(M, f()))) / M
print(pi_est)

## Switch back to sequential processing
plan(sequential)

## Shut down manually launched mirai workers
invisible(mirai::daemons(0))
```

`mirai_multisession` *Mirai-based localhost multisession futures*

Description

WARNING: This function must never be called. It may only be used with `future::plan()`

Usage

```
mirai_multisession(..., workers = availableCores(), envir = parent.frame())
```

Arguments

`workers`	The number of parallel processes to use. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using a sequential future. To override this fallback, use `workers = I(1)`.
`envir`	The environment from where global objects should be identified.
`...`	Not used.

Value

Nothing.

Examples

```
library(future)
plan(future.mirai::mirai_multisession)

# A function that returns a future
# (note that N is a global variable)
f <- function() future({
  4 * sum((runif(N) ^ 2 + runif(N) ^ 2) < 1) / N
}, seed = TRUE)

# Run a simple sampling approximation of pi in parallel using  M * N points:
N <- 1e6  # samples per worker
```

```
M <- 10   # iterations
pi_est <- Reduce(sum, Map(value, replicate(M, f()))) / M
print(pi_est)

## Switch back to sequential processing, which also
## shuts down the automatically launched mirai workers
plan(sequential)
```

Index

environment, *3*, *6*

future.mirai, 2
future.mirai-package *(*future.mirai*)*, 2
future::plan(), *3*, *6*

mirai::cluster_config(), *4, 5*
mirai::daemons(), *4*
mirai_cluster, 3
mirai_multisession, 6
MiraiFutureBackend, 3
MiraiMultisessionFutureBackend
        *(*MiraiFutureBackend*)*, 3