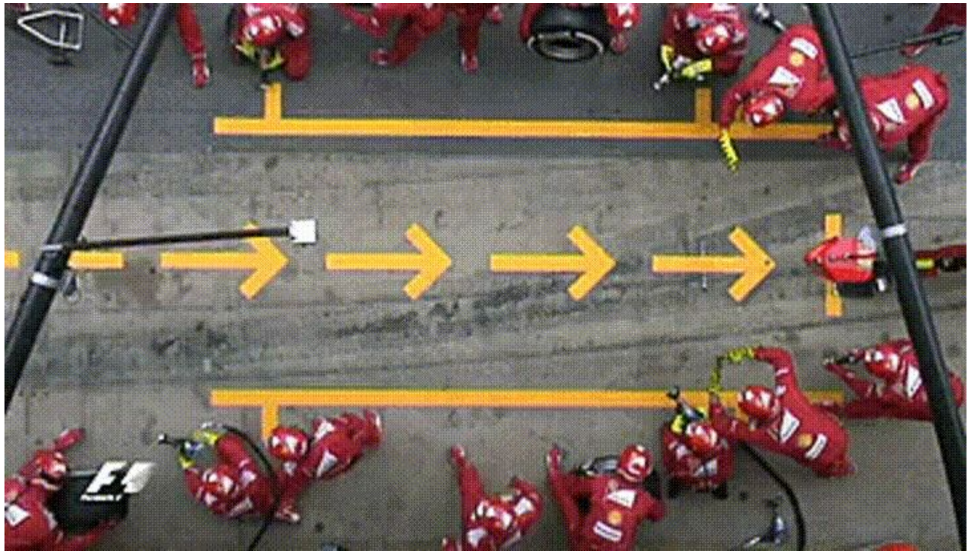
Futureverse: A Unifying Parallelization Framework in R for Everyone - Part 1



Henrik Bengtsson
University of California, San Francisco
R Foundation, R Consortium

@HenrikBengtsson
HenrikBengtsson

RaukR 2024, Visby, Sweden, June 12 & 17, 2024 (https://www.futureverse.org)

We parallelize software for various reasons

Parallel & distributed processing can be used to:

- speed up processing (wall time)
- lower memory footprint (per machine)
- avoid data transfers (compute where data lives)
- Other reasons, e.g. asynchronous UI

We parallelize software for various reasons

We may choose to parallelize on:

- Your personal laptop or work desktop computer (single user)
- A shared powerful computer (multiple users)
- Across many computers, e.g. in the office or in the cloud
- High-performance compute (HPC) cluster (multiple users) with a job scheduler, e.g. Slurm, Son of Grid Engine (SGE)

History - What's Already Available in R?

R comes with built-in parallelization

```
library(DNAseq)
fq <- c("file1.fq", "file2.fq", "file3.fq")   # In: FASTQ files
bam <- lapply(fq, align)                       # 3 hours
## [1] "file1.bam" "file2.bam" "file3.bam"     # Out: BAM files
```

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

```
library(parallel)
bam <- mclapply(fq, align, mc.cores = 3)       # 1 hour
```

To parallelize also on Windows, we can do:

```
library(parallel)
workers <- makeCluster(3)
bam <- parLapply(fq, align, cl = workers)      # 1 hour
```

Things we need to be aware of

mclapply() - magic with problems

Pros:

- `mclapply()` works *similarly* to `lapply()`
- `mclapply()` comes with all R installations
- no need to worry about global variables and loading packages

Cons:

- *Forked* processing ⇒ not supported on MS Windows
- *Forked* processing ⇒ unstable with *multi-threaded* code & GUIs, e.g. may core dump RStudio
- There are no `mcapply(), mcsapply(), mcvapply(), ...`
- Errors have to be handled with exceptionally great care

⚠️ Use forked processing with care!

R Core & `mclapply()` author Simon Urbanek (on R-devel, 2020):

“Do NOT use `mcparallel()` in packages except as a non-default option that user can set ... Multicore is intended for HPC applications that need to use many cores for computing-heavy jobs, but it does not play well with RStudio and more importantly you [as the developer] don't know the resource available so only the user can tell you when it's safe to use.”

parLapply() - takes some efforts

Pros:

- `parLapply()` works just like `lapply()`
- `parLapply()` comes with all R installations
- `parLapply()` works on all operating systems

Cons:

- Requires manually loading of packages on workers, e.g.
  `clusterEvalQ(workers, library(DNAseq))`
- Requires manually exporting globals to workers, e.g.
  `clusterExport(workers, c("varA", "varB"))`
- There are no `parMapply()`, `parVapply()`, …
- Errors have to be handled with great care

Error if we forget to load package on workers

```
library(DNAseq)
align_and_count <- function(fq) {
  bam <- align(fq)
  count_seqs(bam)
}

library(parallel)
workers <- makeCluster(3)

counts <- parLapply(fq, align_and_count, cl = workers)
## Error in checkForRemoteErrors(val) : 3 nodes produced
## errors; first error: could not find function "align"
```

Error if we forget to load package on workers

```
library(DNAseq)
align_and_count <- function(fq) {
  bam <- align(fq)
  count_seqs(bam)
}

library(parallel)
workers <- makeCluster(3)
clusterEvalQ(workers, library(DNAseq))  # <== Don’t forget!

counts <- parLapply(fq, align_and_count, cl = workers)
```

Design patterns found in packages

My “align them all” function

```
align_all <- function(fq) {
  lapply(fq, align)
}
```

```
> fq <- c("file1.fq", "file2.fq", "file3.fq")
> bam <- align_all(fq)
> bam
[1] "file1.bam" "file2.bam" "file3.bam"
```

v1. A first attempt on parallel support

```r
align_all <- function(fq, parallel = FALSE) {
  if (parallel) {
    bam <- mclapply(fq, align, mc.cores = detectCores())
  } else {
    bam <- lapply(fq, align)
  }
  bam
}

> bam <- align_all(fq, parallel = TRUE)
> bam
[1] "file1.bam" "file2.bam" "file3.bam"
```

v2. A much better approach

```
align_all <- function(fq, parallel = FALSE) {
  if (parallel) {
    bam <- mclapply(fq, align) # let user decide on cores!👍
  } else {
    bam <- lapply(fq, align)
  }
  bam
}

> options(mc.cores = 4)
> bam <- align_all(fq, parallel = TRUE)
```

v3. Yet another alternative

```r
align_all <- function(fq, ncores = 1) {
  if (ncores > 1) {
    bam <- mclapply(fq, align, mc.cores = ncores)
  } else {
    bam <- lapply(fq, align)
  }
  bam
}

> bam <- align_all(fq, ncores = 4)
```

v4. Support also MS Windows

```
align_all <- function(fq, ncores = 1) {
  if (ncores > 1) {
    if (.Platform$OS.type == "windows") {
      workers <- makeCluster(ncores)
      on.exit(stopCluster(workers))
      clusterEvalQ(workers, library(somepkg))
      bam <- parLapply(fq, align, cl = workers)
    } else {
      bam <- mclapply(fq, align, mc.cores = ncores)
    }
  } else {
    bam <- lapply(fq, align)
  }
  bam
}
```

More feature requests ...

- *Can you please add support for AAA parallelization too?*
- *While you’re at it, what about BBB parallelization?*

v99: Phew … will this do?

```r
align_all <- function(fq, parallel = "none") {
  if (parallel == "snow") {
    workers <- getDefaultCluster()
    clusterEvalQ(workers, library(somepkg))
    bam <- parLapply(fq, align, cl = workers)
  } else if (parallel == "multicore") {
    bam <- mclapply(fq, align)
  } else if (parallel == "clustermq") {
    bam <- clustermq::Q(align, fq, pkgs="somepkg")
  } else if (parallel == ...) {
    ...
  } else {
    bam <- lapply(fq, align)
  }
  bam
}
```

What's my test coverage now?

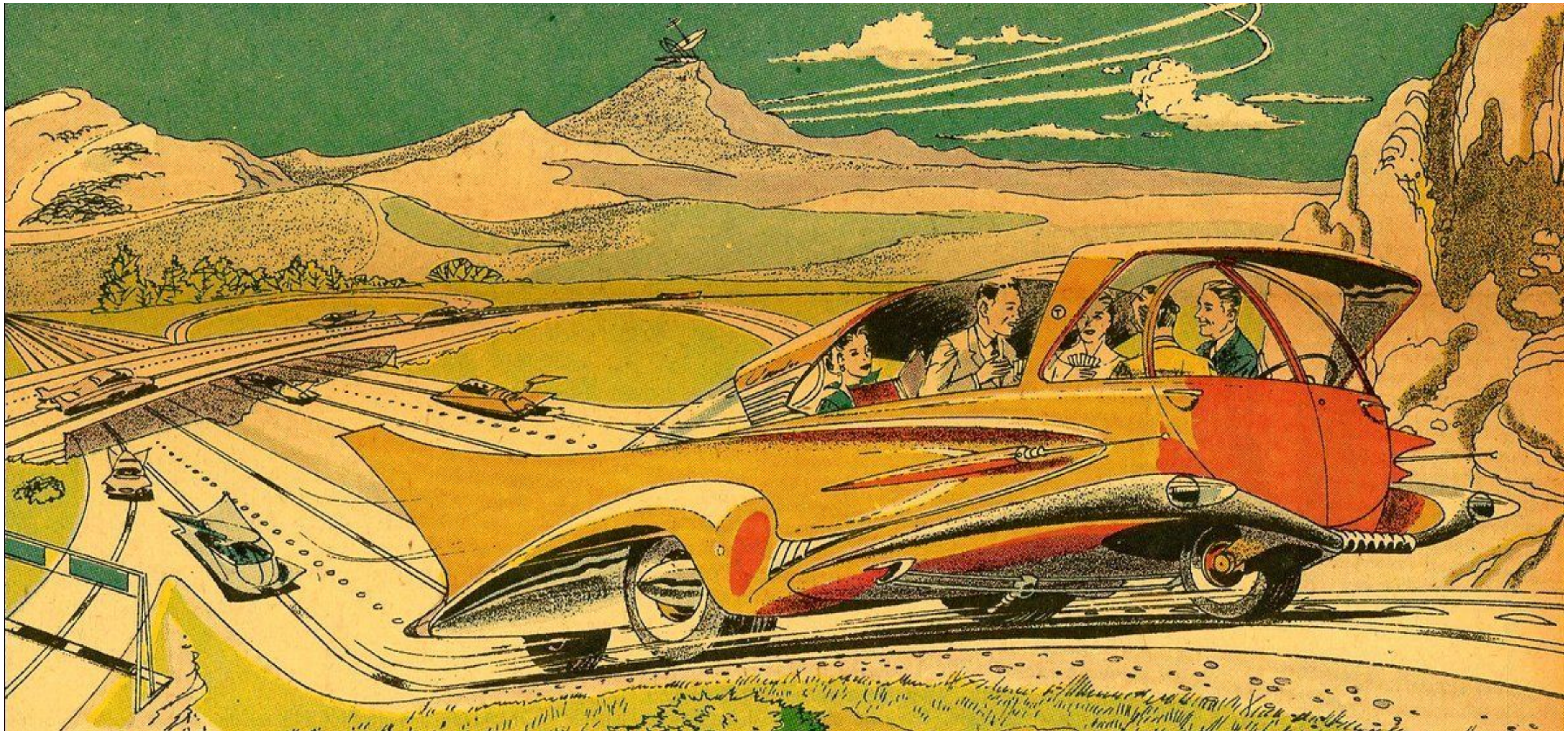
Some months later …

- *There is this new, cool DDD parallelization method … ?*

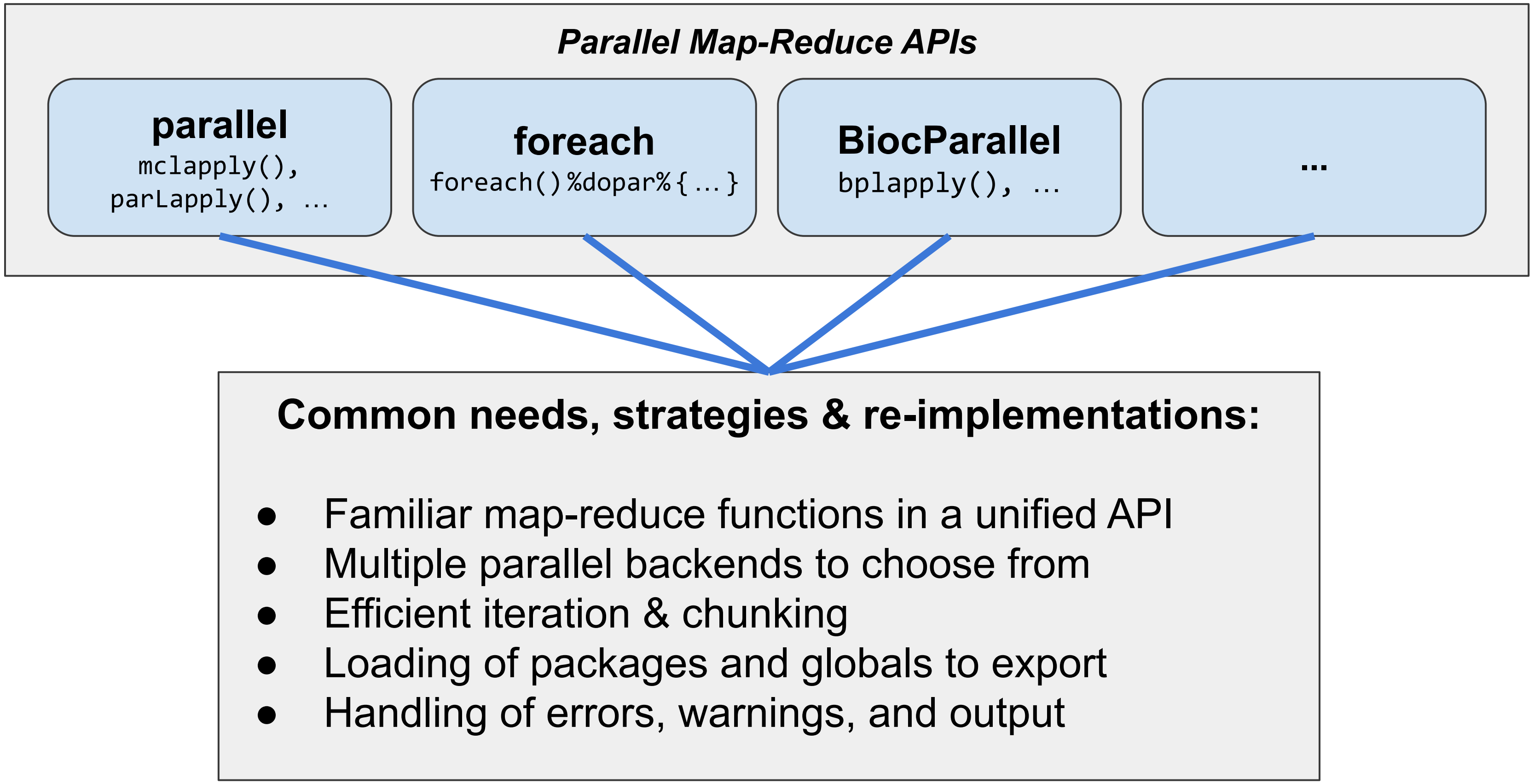
- *…*

- *Still there?*

Welcome to the Future



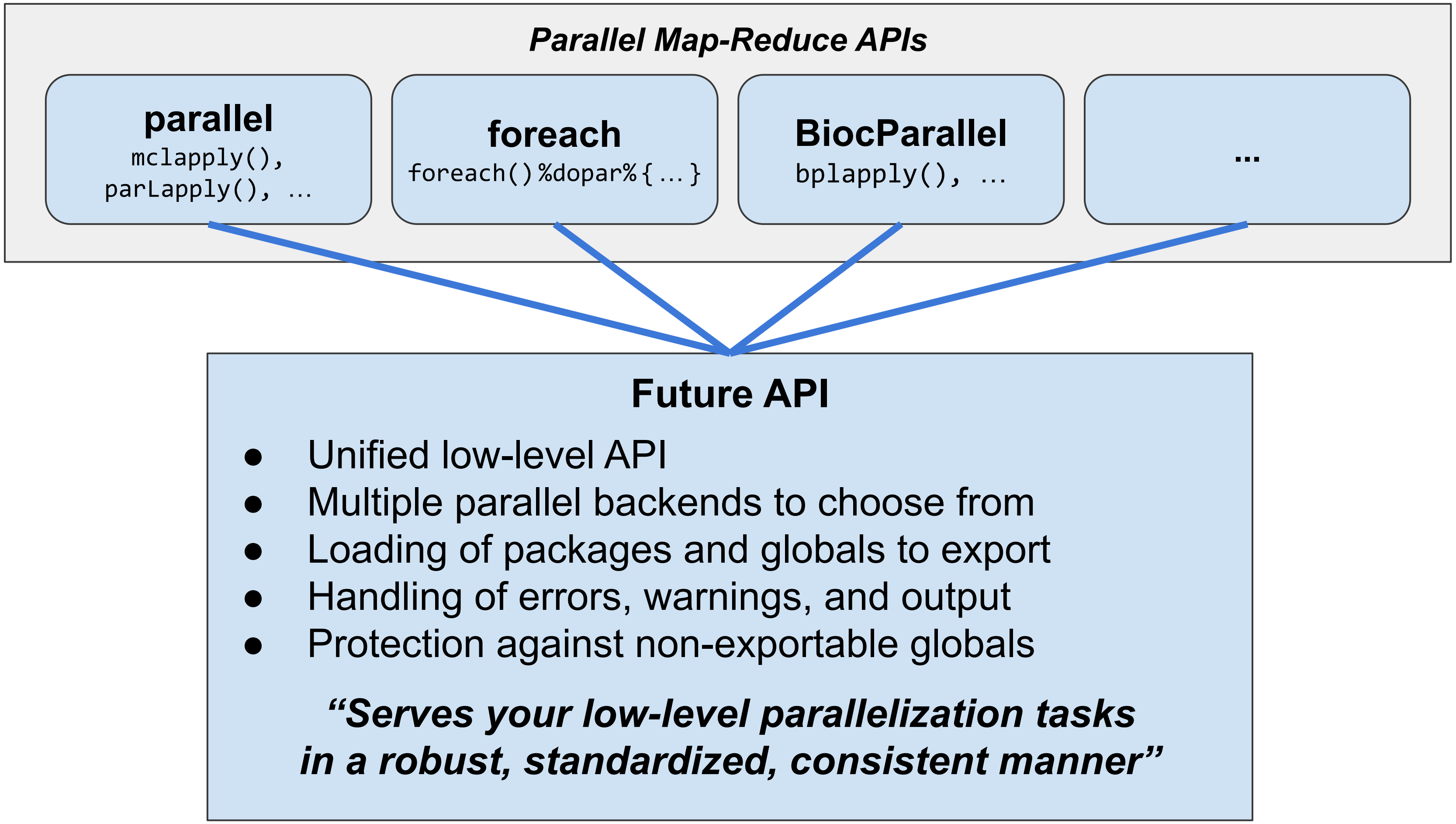
Parallel frameworks reimplement common ideas



Common needs, strategies & re-implementations:

- Familiar map-reduce functions in a unified API
- Multiple parallel backends to choose from
- Efficient iteration & chunking
- Loading of packages and globals to export
- Handling of errors, warnings, and output

Idea: Collect common tasks in one place



Future API

- Unified low-level API
- Multiple parallel backends to choose from
- Loading of packages and globals to export
- Handling of errors, warnings, and output
- Protection against non-exportable globals

"Serves your low-level parallelization tasks in a robust, standardized, consistent manner"

R package: future

- "Write once, run anywhere"
- 100% cross-platform
- Works with any type of parallel backends
- A simple unified API
- Easy to install (< 0.5 MiB total)
- Very well tested, lots of CPU mileage

Dan LaBar
@embiggenData

“Low friction”:

- automatically exports global variables
- automatically relays output, messages, and warnings
- proper parallel random number generation (RNG)

A Future is ...

- A future is an abstraction for a value that will be available later
- The state of a future is either unresolved or resolved
- The value is the result of an evaluated expression

An R assignment:

```
v <- expr
```

Future API:

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

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

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                        # 1275 + 3775
[1] 5050
```

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

```
> library(future)
> plan(multisession)  # parallelize on local computer

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

> a <- value(fa)                      # blocks until ready
> b <- value(fb)
> a + b                               # here at ~1 minute
[1] 5050
```

User chooses how to parallelize - many options

```r
plan(sequential)

plan(multicore)               # uses the mclapply() machinery

plan(multisession)            # uses the parLapply() machinery

plan(cluster, workers = c("n1", "n2", "n3"))

plan(cluster, workers = c("n1", "m2.uni.edu", "vm.cloud.org"))

plan(batchtools_slurm)        # on a Slurm job scheduler

plan(future.callr::callr)     # locally using callr package

plan(future.mirai::mirai_multisession) # locally using mirai package

...
```

Parallelize on other machines is easy

```
> library(future)
> plan(cluster, workers = c("alice", "bob"))

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

> a <- value(fa)                      # blocks until ready
> b <- value(fb)
> a + b                               # here at ~1 minute
[1] 5050
```

Parallelize on other machines is easy

```
> library(future)
> plan(cluster, workers = c("alice", "bob", "carole", "dave"))

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

> a <- value(fa)                      # blocks until ready
> b <- value(fb)
> a + b                               # here at ~1 minute
[1] 5050
```

Parallelize on other machines is easy

```
> library(future)
> plan(cluster, workers = c("alice", "bob", "carole", "dave"))

> fa <- future( slow_sum( 1:25 ) )    # ~0 seconds
> fb <- future( slow_sum(26:50 ) )    # ~0 seconds
> fc <- future( slow_sum(51:75 ) )    # ~0 seconds
> fd <- future( slow_sum(76:100) )    # ~0 seconds

> y <- value(fa) + value(fb) + value(fc) + value(fd)
> y                                    # here at ~30 seconds
[1] 5050
```

Globals automatically identified (99% worry free)

Static-code inspection by walking the abstract syntax tree (AST):

```
x <- rnorm(n = 100)          lobstr::ast( { slow_sum(x) } )
f <- future({ slow_sum(x) })        |  █─`{`
                     |______________|  └─█─slow_sum
                                    |    └─x
```

=> globals & packages identified and exported to the worker:

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

Comment: Globals & packages can also be specified manually;

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

Other frameworks need manual exports

With other parallel frameworks, you have to manually export the globals that need to be available on the parallel workers, e.g.

```
library(parallel)
cl <- makeCluster(2)
x <- rnorm(n = 100)
clusterExport(cl, c("slow_sum", "x"))
y <- clusterEvalQ(cl, { slow_sum(x) })
```

Conclusion: This is *not* needed when using Futureverse for parallelization (except for rare, corner cases)