12-2021

A Unifying Framework for Parallel and Distributed Processing in R using Futures

Henrik Bengtsson

A Unifying Framework for Parallel and Distributed Processing in R using Futures

by Henrik Bengtsson

Abstract A *future* is a programming construct designed for concurrent and asynchronous evaluation of code, making it particularly useful for parallel processing. The **future** package implements the *Future API* for programming with futures in R. This minimal API provides sufficient constructs for implementing parallel versions of well-established, high-level map-reduce APIs. The future ecosystem supports exception handling, output and condition relaying, parallel random number generation, and automatic identification of globals lowering the threshold to parallelize code. The *Future API* bridges parallel frontends with parallel backends, following the philosophy that end-users are the ones who choose the parallel backend while the developer focuses on what to parallelize. A variety of backends exist, and third-party contributions meeting the specifications, which ensure that the same code works on all backends, are automatically supported. The future framework solves several problems not addressed by other parallel frameworks in R.

Introduction

Parallel processing can be used to speed up computationally intensive tasks. As the size of these tasks and access to more CPU cores tend to grow over time, so does the demand for parallel-processing solutions. In R, there exist several frameworks for running code in parallel, many dating back more than a decade (Schmidberger et al., 2009). R gained built-in support via the **parallel** package in version 2.14.0 (2011), which to date probably provides the most, either directly or indirectly, commonly used solutions. For an overview of current parallel techniques available to R developers, see Eddelbuettel (2021) and the *High-Performance and Parallel Computing with R* CRAN Task View.

The options for parallelizing *computations* in R can be grouped broadly into those that can be used to parallelize R code, such as what the **parallel** package provides, and those that are used to parallelize native code, such as C, C++, and Fortran, and are often not specific to R itself. For example, multi-threaded processing is an efficient parallelization technique which operates at the core of the operating system and the CPU and allows for updating shared memory in parallel and more, which is not available at the R level. In contrast, parallelization at the R level takes place at a higher level with a coarser type of parallelization, which we refer to as *multi-process* parallelization. In addition to parallel computations, there are also efforts in R for working with *parallel data structures*, e.g., **sparklyr** (Luraschi et al., 2021) and the *Programming with Big Data in R* (pbdR) project (Schmidt et al., 2017). By pre-distributing data and storing them on, or near, parallel workers, the overhead from passing data on-the-fly in parallel processing can be decreased, resulting in an overall faster processing time but also lower and more fine-tuned memory requirements. This article proposes a solution for *parallelizing computations at the R level*.

The **future** package (Bengtsson, 2021b) aims to provide a unifying, generic, minimal application protocol interface (API) to facilitate the most common types of parallel processing in R, especially the *manager-worker* strategy where an R process delegates tasks to other R processes. It builds upon the concepts of *futures* (Hewitt and Baker, 1977) and *promises* (Friedman and Wise, 1978; Hibbard, 1976) - concepts that are well suited for a functional language such as R. To better understand how it fits in among and relates to existing parallelization solutions in R[1], let us revisit the two most well-known solutions - packages **parallel** and **foreach**.

The **parallel** package has a set of functions for calling functions and expressions in parallel across one or more concurrent R processes. The most well-known functions for this are `mclapply()` and `parLapply()`, which mimic the behavior of the map-reduce[2] function `lapply()` in the **base** package. Below is an example showing them calling a "slow" function on each element in a vector using two parallel workers. First, to do this through sequentially processing, we can use `lapply()`:

```
xs <- 1:10
y <- lapply(xs, function(x) {
  slow_fcn(x)
})
```

[1]Although the concept of futures could also apply to C, C++, and Fortran parallelization, the future framework targets parallelization at the R level and does not provide an implementation for native code.

[2]We use the term "map-reduce" as it is used in functional programming. The *MapReduce* method by Dean and Ghemawat (2004) was inspired by this term but they are not equivalent.

To do the same in parallel using two *forked* parallel processes, we can use:

```
library(parallel)
xs <- 1:10
y <- mclapply(xs, function(x) {
  slow_fcn(x)
}, mc.cores = 2)
```

Alternatively, to run it in parallel using two R parallel processes running in the *background*, we can do:

```
library(parallel)
workers <- makeCluster(2)
clusterExport(workers, "slow_fcn")
xs <- 1:10
y <- parLapply(workers, xs, function(x) {
  slow_fcn(x)
})
```

These functions, which originate from legacy packages **multicore** (2009-2014, Urbanek (2014)) and **snow** (since 2003, Tierney et al. (2021)), are designed for specific parallelization frameworks. The `mclapply()` set of functions relies on process *forking* by the operating system, which makes them particularly easy to use. This is because each worker automatically inherits the setup and all of the content of the main R process' workspace, making it straightforward to replace a sequential `lapply()` call with a parallel `mclapply()` call. This has made it popular among Linux and macOS developers. On MS Windows, where R does not support forked processing, `mclapply()` falls back to using `lapply()` internally.

The `parLapply()` set of functions, which all operating systems support, rely on a cluster of R background workers for parallelization. It works by the main R process and the workers exchanging tasks and results over a communication channel. The default and most commonly used type of cluster is SOCK, which MS Windows also supports, and it communicates via *socket connections*. Like most other cluster types, SOCK clusters require developers to manually identify and export packages and global objects to the workers by calling `clusterEvalQ()` and `clusterExport()`, before calling `parLapply()`, which increases the barrier to use them.

Mixed responsibilities of developers or end-users

Using either the `mclapply()` or the `parLapply()` approach works well when developers and end-users can agree on which framework to use. Unfortunately, this is not always possible, e.g., R package developers rarely know who the end-users are and what compute resources they have. Regardless, developers who wish to support parallel processing still face the problem of deciding which parallel framework to target, a decision that often has to be done early in the development cycle. This means deciding on what *type of parallelism* to support, e.g., forked processing via `mclapply()` or SOCK clusters via `parLapply()`. This decision is critical because it limits the end-user's options, and any change, later on, might be expensive because of, for instance, having to rewrite and retest part of the codebase. A developer who wishes to support multiple parallel backends has to implement support for each of them individually and provide the end-user with a mechanism to choose between them. This approach often results in unwieldy, hard-to-maintain code of conditional statements with low test coverage, e.g.,

```
if (parallel == "fork") {
  ...
} else if (parallel == "SOCK") {
  ...
} else if (parallel == "MPI") {
  ...
} else {
  ...
}
```

There is no established standard for doing this, which results in different packages providing different mechanisms for controlling the parallelization method, if at all.

Functions like `parLapply()` partly address the problem of supporting multiple parallelization frameworks because they support various types of parallel cluster backends referred to as "snow" clusters (short for *Simple Network of Workstations* and from their origin in the **snow** package), e.g., `workers <- makeCluster(4, type = "FORK")` sets up a cluster that parallelizes using forked processing,

and `workers <- makeCluster(4, type = "MPI")` sets up a cluster that parallelizes via a Message Passing Interface (MPI) framework. If a developer uses `parLapply()`, they could write their code such that the end-user can specify what type of snow cluster to use, e.g., by respecting what the end-user set via `setDefaultCluster(workers)`. This provides the end-user with more, although in practice limited, options on how and where to execute code in parallel. Unfortunately, it is rather common that the cluster type is hard-coded inside packages giving end-users little to no control over the parallelization mechanism, other than possibly the number of cores to use.

Map-reduce parallelization with more control for the end-user

Possibly inspired by the snow-style clusters, the **foreach** package (Microsoft and Weston, 2020; Kane et al., 2013), first released in 2009, addresses the above problem of having to decide on the parallel design early on by letting the end-user - not the developer - "register" what type of parallel backend ("foreach adaptor") to use when calling `foreach()`. For example, with **doMC** (Revolution Analytics and Weston, 2020), one can register a multicore cluster, and with **doParallel** (Microsoft Corporation and Weston, 2020), one can register any type of "snow" cluster as in:

```
library(foreach)
library(doParallel)
workers <- parallel::makeCluster(2)
registerDoParallel(workers)

xs <- 1:10
y <- foreach(x = xs) %dopar% {
  slow_fcn(x)
}
```

We note that the specification of what type of parallel framework and number of cores to use is separated from the `foreach()` map-reduce construct itself. This gives more control to the end-user on *how* and *where* to parallelize, leaving the developer to focus on *what* to parallelize, which is a design pattern of great value with important implications on how to design, write, and maintain parallel code. The large uptake of **foreach** since it was first released supports this. As of November 2021, **foreach** is among the top-1.0% most downloaded packages on CRAN, and there are 867 packages on CRAN and Bioconductor that directly depend on it. Another advantage of the separation between the map-reduce frontend API and parallel backend (foreach adaptors) is that new types of parallel backends can be introduced without the need to make updates to the **foreach** package. This has led to third-party developers have contributed additional foreach adaptors, e.g., **doMPI** (Weston, 2017) and **doRedis** (Lewis, 2020).

Unfortunately, there is no *exact* specification on what a foreach adaptor should support and how it should act in certain situations, which has resulted in adaptors behaving slightly differently. At their face value, these differences appear innocent but may cause different outcomes of the same code. In the best case, these differences result in run-time errors, and in the worst case, different results. An example of the former is the difference between **doMC** on Unix-like systems and **doParallel** on Windows. Analogously to `mclapply()`, when using **doMC**, globals and packages are automatically taken care of by the process forking. In contrast, when using **doParallel** with "snow" clusters, globals and packages need to be identified and explicitly exported, via additional arguments `.export` and `.packages` to `foreach()`, to the parallel workers running in the background. Thus, a developer that only uses **doMC** might forget to test their code with **doParallel**, where it may fail. Having said this, the **foreach** package does provide a rudimentary mechanism for automatically identifying and exporting global variables. However, it has some limitations, that, in practice, require the developer to explicitly specify globals to make sure their code works with more backends. Some adaptors provide additional options of their own that are specified as arguments to `foreach()`. If the developer specifies such options, the `foreach()` call might not work with other adaptors.

To develop `foreach()` code invariant to the parallel backend chosen requires a good understanding of how the **foreach** framework works and plenty of testing. This lack of strict behavior is unfortunate and might have grown out of a strategy of wanting to keep things flexible. On the upside, steps have recently[3] been taken toward making the behavior more consistent across foreach backends, suggesting that it is possible to remove several of these weaknesses through a process of deprecating and removing unwanted side effects over several release cycles in close collaboration with package developers currently relying on such backend-specific properties.

[3]See the **foreach** issue tracker at https://github.com/RevolutionAnalytics/foreach.

The future framework

The **future** package defines and implements the *Future API* - a minimal, unifying, low-level API for parallel processing, and more. Contrary to the aforementioned solutions, this package does *not* offer a parallel map-reduce API per se. Instead, it focuses on providing efficient and simple-to-use atomic building blocks that allow us to implement such higher-level functions elsewhere.

Three atomic constructs that unify common parallel design patterns

The *Future API* comprises three fundamental constructs:

- `f <- future(expr)` : evaluates an expression via a future (non-blocking, if possible)
- `v <- value(f)` : the value of the future expression `expr` (blocking until resolved)
- `r <- resolved(f)` : TRUE if future is resolved, otherwise FALSE (non-blocking)

To help understand what a future is, let us start with R's assignment construct:

```
v <- expr
```

Although it is effectively a single operator, there are two steps in an assignment: first (i) R evaluates the *expression* on the right-hand side (RHS), and then (ii) it assigns the resulting value to the *variable* on the left-hand side (LHS). We can think of the *Future API* as giving us full access to these two steps by rewriting the assignment construct as:

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

Contrary to the regular assignment construct where the evaluation of the expression and the assignment of its value are tightly coupled, the future construct allows us to decouple these steps, which is an essential property of futures and necessary when doing parallel processing[4]. Especially, the decoupling allows us to perform other tasks in-between the step that evaluates the expression and the step that assigns its value to the target variable. Here is an example that creates a future that calculates `slow_fcn(x)` with `x` being `1`, then reassigns a different value to `x`, and finally gets the value of the future expression:

```
x <- 1
f <- future({
  slow_fcn(x)
})
x <- 2
v <- value(f)
```

By definition, a future consists of an R expression and any required objects as they were when the future was created. Above, the recorded objects are the function `slow_fcn()` and the variable `x` with value `1`. This is why the value of the future is unaffected by `x` getting reassigned a new value after the future is created but before the value is collected.

We have yet to explain how futures are resolved, that is, how the future expression is evaluated. This is the part where futures naturally extend themselves to asynchronous and parallel processing. How a future is resolved depends on what *future backend* is set. If not specified, the default is to resolve futures sequentially, which corresponds to setting:

```
plan(sequential)
```

Before we continue, it should be emphasized that the *Future API* is designed so that a program using it gives the same results no matter how and where the futures are resolved, may it be sequentially on the local machine or in parallel on a remote cluster. As a consequence, *the future ecosystem is designed to separate the responsibilities of the developer from those of the end-user*. This allows the developer to focus on the code to be parallelized while the end-user focuses on how to parallelize. It is the end-user who decides on the `plan()`. For example, if they specify:

```
plan(multisession)
```

[4]We can find this future-value pattern in several implementations for parallel processing, including the ones we use in R. The `mcparallel()`-`mccollect()` pair of functions in **parallel** is one example. This is why the future-value abstraction can be mapped onto many of our existing parallel frameworks in a unified way.

before calling the above future code, futures will be resolved in parallel via a SOCK cluster on the local machine similar to what we used above in the `parLapply()` example. If the end-user instead specifies `plan(multicore)`, futures will be resolved in parallel in the background via *forked* R processes using the same framework as `mclapply()`. Importantly, regardless of what future plan is used, and regardless of whether or not we assigned a new value to `x` after creating the future, the result is always the same. Since we, as developers, do not know what backend end-users will use, we also cannot know *when* a future is resolved. This is why we say that "a future evaluates its expression *at some point in the future*". What we do know is that `value()` returns the value of the future only when it is resolved, and if it is not resolved, then `value()` waits until it is.

Next, let us look at how blocking works by using an example where we create three futures to be resolved by two parallel workers:

```
library(future)
plan(multisession, workers = 2)

xs <- 1:10

f1 <- future({
  slow_fcn(xs[1])
})

f2 <- future({
  slow_fcn(xs[2])
})

f3 <- future({
  slow_fcn(xs[3])
})
```

Here, the first two futures are created in a non-blocking way because there are two workers available to resolve them. However, when we attempt to create a third future, there are no more workers available. This causes `future()` to block until one of the workers is available, that is, until either one or both of the two futures have been resolved. If three or more workers are set up, then the third `future()` call would not block. On the other hand, if `plan(sequential)` is set, then each `future()` blocks until the previously created future has been resolved. Finally, to retrieve the values of the three futures, we do:

```
v1 <- value(f1)
v2 <- value(f2)
v3 <- value(f3)
```

Although it is common to call `value()` on the futures in the order we created them, we can collect the values in any order, which is something we will return to later.

Continuing, we can generalize the above to calculate `slow_fcn()` on each of the elements in `xs` via futures. For this, we can use a regular for-loop to create each of the `length(xs)` futures:

```
xs <- 1:10
fs <- list()
for (i in seq_along(xs)) {
  fs[[i]] <- future(slow_fcn(xs[i]))
}
```

Note how we here have effectively created a *parallel for-loop*, where `plan()` controls the amount of parallelization. To collect the values of these futures, we can use[5]:

```
vs <- lapply(fs, value)
```

Alternatively, to using a for-loop, we can parallelize using `lapply()`:

```
xs <- 1:10
fs <- lapply(xs, function(x) {
  future(slow_fcn(x))
})
```

This is illustrated in Figure 1, where four background workers created by `plan(multisession,workers = 4)` is used to resolve the futures. The same idea also applies to other types of map-reduce functions.

[5]Here, `vs <- lapply(fs, value)` is used for clarification but we could also have used `vs <- value(fs)` because `value()` is a generic function with implementation also for lists and other types of containers.

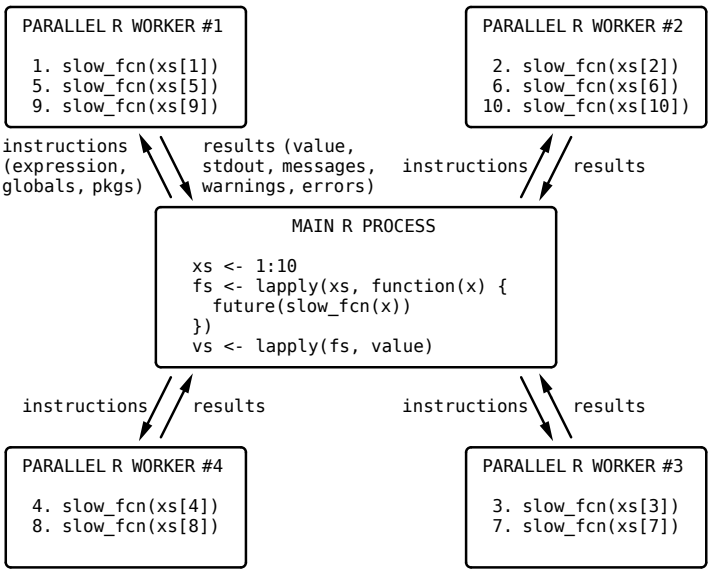

Figure 1: An illustration of parallel processing using futures via four R processes running in the background. Base R `lapply()` is used to call `slow_fcn()` ten times - once per element in `xs`. By calling it via `future()`, each call is distributed out to one of four workers. If all workers are busy, the next future, in turn, will wait for a worker to become available. The results of all futures are collected at the end. Any output, warnings, and errors produced on the workers are relayed as-is back on the main R session. The four workers were created using `plan(multisession, workers = 4)`. If switching to `plan(sequential)`, then all futures are resolved sequentially in the main R process. Only core Future API functions from the **future** package were used. Less verbose, map-reduce alternatives are available in the high-level future packages such as **future.apply**, **furrr**, and **doFuture**.

This shows how powerful the *Future API* is; by combining base R with the two constructs `future()` and `value()`, we have created rudimentary[6] alternatives to `mclapply()`, `parLapply()`, and `foreach()`. Indeed, we could reimplemented these **parallel** and **foreach** functions using the *Future API*.

The `resolved()` function queries, in a non-blocking way, whether or not a future is resolved. Among other things, this can be used to collect the value of a subset of resolved futures as soon as possible without risking to block from collecting the value of a non-resolved future, which allows additional futures to launch sooner, if they exist. This strategy also helps lower the overall latency that comes from the overhead of collecting values from futures - values that may contain large objects and are collected from remote machines over a network with limited bandwidth. As explained further below, collecting the value of futures as soon as possible will also lower the latency of the relay of output and conditions (e.g., warnings and errors) captured by each future while they evaluate the future expressions.

In summary, the three constructs of the *Future API* provide *the necessary and sufficient* functionality for evaluating R expressions in parallel, which in turn may be used to construct higher-level map-reduce functions for parallel processing. Additional core features of futures that are useful, or even essential, for parallel processing are presented next.

Exception handling

To make it as simple as possible to use futures, they are designed to mimic the behavior of the corresponding code that does not use futures. An important part of this design aim is how exception handling is done. Any *error* produced while resolving a future, that is, evaluating its expression, is captured and relayed as-is in the main R process each time `value()` is called. This mimics the behavior of how errors are produced when not using futures. This is illustrated by the following two code examples – with futures:

[6]These solutions process each element in a separate future, which is suboptimal if the overhead of creating a future is relatively large compared to the evaluation time. This overhead can be mitigated by processing elements in chunks, something that requires more complex code than what is shown in these examples.

```
x <- "24"
f <- future(log(x))
v <- value(f)
# Error in log(x) : non-numeric argument to mathematical function
```

and without futures:

```
x <- "24"
v <- log(x)
# Error in log(x) : non-numeric argument to mathematical function
```

As a result, standard mechanisms for condition handling also apply to errors relayed by futures. For example, to assign a missing value to v whenever there is an error, we can use:

```
v <- tryCatch({
  value(f)
}, error = function(e) {
  NA_real_
})
```

Errors due to extraordinary circumstances, such as terminated R workers and failed communication, are of a different kind than the above evaluation errors. Because of this, they are signaled as errors of class *FutureError* so they can be handled specifically, e.g., by restarting R workers or relaunching the failed future elsewhere (Section 'Future work').

Relaying of standard output and conditions (e.g., messages and warnings)

Futures capture the standard output (*stdout*) and then relay it in the main R process each time `value()` is called. Analogously, all conditions are captured and relayed as-is in the main R process each time `value()` is called. Common conditions relayed this way are *message*s and *warning*s as generated by `message()` and `warning()`. The relaying of errors was discussed in the previous section. Relaying of standard output and conditions respects the order they were captured, except that all of the standard output is relayed before conditions are relayed in the order they were signaled. For example,

```
x <- c(1:10, NA)
f <- future({
  cat("Hello world\n")
  y <- sum(x, na.rm = TRUE)
  message("The sum of 'x' is ", y)
  if (anyNA(x)) warning("Missing values were omitted", call. = FALSE)
  cat("Bye bye\n")
  y
})
v <- value(f)
# Hello world
# Bye bye
# The sum of 'x' is 55
# Warning message:
# Missing values were omitted
```

Standard techniques can be used to capture the relayed standard output, e.g.,

```
stdout <- capture.output({
  v <- value(f)
})
# The sum of 'x' is 55
# Warning message:
# Missing values were omitted

stdout
# [1] "Hello world" "Bye bye"
```

Similarly, `withCallingHandlers()` and `globalCallingHandlers()` can be used to capture and handle the different classes of conditions being relayed. Note that all of the above works the same way regardless of what future backend is used, including when futures are resolved on a remote machine.

Relaying of standard output, messages, warnings, and errors simplifies any troubleshooting. For example, existing verbose output helps narrow down the location of errors and warnings, which may

reveal unexpected missing values or vector recycling. Commonly used poor-man debugging, where temporary debug messages are injected into the code, is also possible because of this built-in relay mechanism. Imagine a logging framework that leverages R's condition framework to signal different levels of log events and then captures and reports, e.g., to the terminal or to file. It will work out of the box when parallelizing with futures.

Conditions of class *immediateCondition* are treated specially by the future framework. They are by design allowed to be relayed as soon as possible, and not only when `value()` is called. For instance, they may be relayed when calling `resolved()`, or even sooner, depending on the future backend used. Because of this, *immediateCondition* conditions are relayed without respecting the order of other types of conditions captured. This makes them suitable for signaling, for instance, *progress updates*. Thus, such progress conditions can be used to update a progress bar in the terminal or in a Shiny application while originating from futures being resolve on remote machines. See the **progressr** package (Bengtsson, 2021h) for an implementation of this. Note, however, that this type of near-live relaying of *immediateCondition*s only works for backends that have the means to communicate these conditions from the worker back to the main R session, while the worker still processes the future. When non-supporting backends are used, these conditions are relayed together with other captured conditions at the very end when the future has been resolved.

Comment: Contrary to the standard output, due to limitations in R[7], it is not possible to capture the standard error reliably. Because of this, any output to the standard error is silently ignored, e.g., `cat("some output", file = stderr())`. However, although output from `message()` is sent to the standard error, it is indeed outputted in the main R processes because it is the message conditions that are captured and relayed, not the standard error.

Globals and packages

The future framework is designed to make it as simple as possible to implement parallel code. Another example of this is the automatic identification of *globals* - short for global variables and functions - that are required for a future expression to be resolved successfully. For example, in:

```
f <- future({
  slow_fcn(x)
})
```

the globals of the future expression are `slow_fcn()` and `x`. By default, `future()` will attempt to identify, locate, and record these globals internally via static code inspection, such that they are available when the future is resolved. If one of these globals is part of a package namespace, that is also recorded. Because of this, developers rarely need to worry about globals when programming with futures. However, occasionally, the future expression is such that it is not possible to infer all the globals. For example, the following produces an error:

```
plan(multisession)
k <- 42
f <- future({
  get("k")
})
v <- value(f)
# Error in get("k") : object 'k' not found
```

This is because code inspection cannot infer that `k` is a needed variable. In such cases, one can guide the future framework to identify this missing global by explicitly mentioning it at the top of the future expression, e.g.,

```
f <- future({
  k
  get("k")
})
```

Alternatively, one can specify it via argument `globals` when creating the future, e.g.,

```
f <- future({
  get("k")
}, globals = "k")
```

See `help("future", package = "future")` for all options available to control which globals to use and how to ignore false positives.

[7] See https://github.com/HenrikBengtsson/Wishlist-for-R/issues/55

Internally, the future framework uses **globals** (Bengtsson, 2020), and indirectly **codetools** (Tierney, 2020), to identify globals by walking the abstract syntax tree (AST) of the future expression in order. It uses an *optimistic* search strategy to allow for some false-positive globals to minimize the number of false-negative globals. Contrary to false positives, false negatives cause futures to produce errors similar to the one above.

Proper parallel random number generation

The ability to produce high-quality random numbers is essential for the validity of many statistical analyses, e.g., bootstrap, permutation tests, and simulation studies. R has functions at its core for drawing random numbers from common distributions. This R functionality is also available to C and Fortran native code. All draw from the same internal pseudo-random number generator (RNG). Different kinds of RNGs are available, with Mersenne-Twister (Matsumoto and Nishimura, 1998) being the default. Like most other RNGs, the Mersenne-Twister RNG is not designed for concurrent processing - if used in parallel, one risks producing random numbers that are correlated. Instead, for parallel processing, the multiple-recursive generator L'Ecuyer-CMRG by L'Ecuyer (1999), implemented in the **parallel** package, can be used to set up multiple RNG streams. The future ecosystem has built-in support for L'Ecuyer-CMRG at its core to make it as easy as possible to produce statistically sound and reproducible random numbers regardless of how and where futures are resolved, e.g.,

```
f <- future(rnorm(3), seed = TRUE)
value(f)
#  [1] -0.02648871 -1.73240257  0.78139056
```

Above, `seed = TRUE` is used to specify that parallel RNG streams should be used. When used, the result will be fully reproducible regardless of future backend specified and the number of workers available. Because `seed = TRUE` can introduce significant overhead, the default is `seed = FALSE`. However, since it is computationally cheap to detect when a future expression produced random numbers, the future framework will generate an informative warning when this is used by mistake to help lower the risk of producing statistically questionable results. It is possible to disable this check or to escalate the warning to an error via an R option. All higher-level parallelization APIs that build upon futures must adhere to this parallel-RNG design, e.g., **future.apply** and **furrr**.

Future assignment construct

As an alternative for using `future()` and `value()`, the **future** package provides a *future-assignment operator*, `%<-%`, for convenience. It is designed to mimic the regular assignment operator, `<-`, in R:

```
v <- expr
```

By replacing the above with:

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

the RHS expression `expr` will be evaluated using a future whose value is assigned to the LHS variable `v` as a *promise*[8]. Because the LHS is a promise, the value of the future will not be assigned to it until we attempt to access the promise. As soon as we try to use `v`, say,

```
y <- sqrt(v)
```

the associated promise will call `value()` on the underlying future, while possibly blocking, and at the end assign the collected result to `v`[9]. From there on, `v` is a regular value. As an illustration, our introductory example with three futures can be written as[10]:

```
xs <- 1:10
v1 %<-% slow_fcn(xs[1])
v2 %<-% slow_fcn(xs[2])
v3 %<-% slow_fcn(xs[3])
```

and with, say, `plan(multisession)`, these statements will be processed in parallel.

Special *infix operators* are available to specify arguments that otherwise would be passed to the `future()` function. For example, to set `seed = TRUE`, we can use:

[8]The type of promises that R supports should not be mistaken for the type of promises as defined by the **promises** (Cheng, 2021) package, which, together with futures, is used for asynchronous processing in Shiny applications.

[9]The internal call to `value()` will also cause any captured standard output and conditions to be relayed.

[10]I have dropped the curly brackets on the RHS to make the example tidier. Just like with regular assignment, there is nothing preventing us from using composite expressions also with future assignments.

```
v %<-% rnorm(3) %seed% TRUE
```

See `help("%<-%", package = "future")` for other infix operators.

Regular R assignments can often be replaced by future assignments as-is. However, because future assignments rely on promises, and promises can only be assigned to *environment*s, including the working environment, they cannot be used to assign to, for instance, *list*s. As a workaround, one can use a *list environment* instead of a *list*. They are implemented in the **listenv** package (Bengtsson, 2019). A list environment is technically an *environment* that emulates most properties of a *list*, including indexing as in:

```
xs <- 1:10
vs <- listenv::listenv()
for (i in seq_along(xs)) {
  vs[[i]] %<-% slow_fcn(xs[i])
}
vs <- as.list(vs)
```

Nested parallelism and protection against it

A problem with parallel processing in software stacks like the R package hierarchy is the risk of overloading the CPU cores due to nested parallelism. For instance, assume that package **PkgA** calls `PkgB::estimate()` in parallel using all N cores on the current machine. Initially, the `estimate()` function was implemented to run sequentially, but, in a recent **PkgB** release, it was updated to parallelize internally using all N cores. Without built-in protection, this update now risks running N^2 parallel workers when **PkgA** is used, possibly without the awareness of either maintainer.

The **future** package has built-in protection against nested parallelism. This works by configuring each worker to run in sequential mode unless nested parallelism is explicitly configured. This is achieved by setting options and environment variables that are known to control parallelism in R, e.g., `options(mc.cores = 1)`. Because of this, if **PkgA** and **PkgB** parallelize using the future framework, the nested parallelism above will run with a total of N cores, not N^2 cores. This will also be true for non-future code that respects such settings, e.g., when **PkgB** uses `parallel::mclapply()` with the default `mc.cores` argument.

Nested parallelism can be configured by the end-user via `plan()`. For example, to use two workers for the first layer of parallelization and three for the second, use:

```
plan(list(
  tweak(multisession, workers = 2),
  tweak(multisession, workers = 3)
))
```

This will run at most $2 \times 3 = 6$ tasks in parallel on the local machine. Any nested parallelism beyond these two layers will be processed in sequential mode. That is, `plan(sequential)` is implicit if not specified. When argument `workers` is not specified, it defaults to `parallelly::availableCores()`, which respect a large number of environment variables and R options specifying the number of cores. Because of this, and due to the built-in protection against nested parallelism, using `plan(list(multisession, multisession))` effectively equals using `plan(list(multisession, sequential))`.

A more common scenario of nested parallelism is when we submit tasks to a job scheduler on a compute cluster where each job is allowed to run on multiple cores allotted by the scheduler. As clarified later, this may be configured as:

```
plan(list(
  future.batchtools::batchtools_sge,
  multisession
))
```

where the default `workers = availableCores()` assures that the number of multisession workers used respects what the scheduler assigns to each job.

Future backends

In addition to implementing the *Future API*, the **future** package also implements a set of future backends that are based on the **parallel** package. If no backend is specified, the default is:

```
plan(sequential)
```

which makes all futures to be resolved sequentially in the current R session. To resolve futures in parallel on a SOCK cluster on the local machine, use one of:

```
plan(multisession) ## defaults to workers = availableCores()
plan(multisession, workers = 4)
```

Similarly, to resolving futures in parallel on the local machine via *forked* processing, use one of:

```
plan(multicore) ## defaults to workers = availableCores()
plan(multicore, workers = 4)
```

To resolve futures via *any* type of "snow" cluster, use the `cluster` backend. For example, to use a traditional SOCK cluster or an MPI cluster, use either of:

```
workers <- parallel::makeCluster(4)
plan(cluster, workers = workers)

workers <- parallel::makeCluster(4, type = "MPI")
plan(cluster, workers = workers)
```

To use a SOCK cluster with two remote workers, use:

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

which is short for:

```
workers <- parallelly::makeClusterPSOCK(c("n1.remote.org", "n2.remote.org"))
plan(cluster, workers = workers)
```

This works as long as there is password-less SSH access to these remote machines and they have R installed. Contrary to `parallel::makePSOCKcluster()`, `parallelly::makeClusterPSOCK()` uses reverse-tunneling techniques, which avoids having to configure inward-facing port-forwarding in firewalls, something that requires administrative rights.

Third-party future backends

Besides these built-in future backends, other R packages available on CRAN implement additional backends. As long as these backends conform to the *Future API* specifications, as discussed in Section 'Validation', they can be used as alternatives to the built-in backends. For example, the **future.callr** package (Bengtsson, 2021e) implements a future backend that resolves futures in parallel on the local machine via R processes[11], orchestrated by the **callr** (Csárdi and Chang, 2021) package, e.g.,

```
plan(future.callr::callr)  ## defaults to workers = availableCores()
plan(future.callr::callr, workers = 4)
```

Another example is **future.batchtools** (Bengtsson, 2021d), which implements several types of backends on top of the **batchtools** (Lang et al., 2017) package. Most notably, it provides backends that resolve futures distributed on high-performance compute (HPC) environments by submitting the futures as jobs to a job scheduler, e.g., Slurm, SGE, and Torque/PBS:

```
plan(future.batchtools::batchtools_slurm)
plan(future.batchtools::batchtools_sge)
plan(future.batchtools::batchtools_torque)
```

Yet another example is the **googleComputeEngineR** package (Edmondson, 2019), which provides a "snow" cluster type that supports[12] resolving futures in the cloud on the Google Compute Engine platform.

[11] The callr backend performs similarly to the PSOCK-based multisession backend. However, in contrast to the latter, it does not rely on socket connections, which on MS Windows may require administrative rights on the machine's firewall in order to allow the R process to communicate on certain ports. Moreover, on machines with a large number of cores, PSOCK clusters are limited to 125 parallel workers because that is the maximum number of connections R can have open simultaneously.

[12] It also supports using `parLapply()` functions.

Implementation

The future framework is platform-independent and works on all platforms, including Linux, Solaris, macOS, and MS Windows. It is backward compatible with older versions of R back to R 3.1.2 (October 2014). The core packages **future**, **parallelly** (Bengtsson, 2021g), **globals**, and **listenv** are implemented in plain R (without native code) to maximize cross-platform operability and to keep installation simple. They are available on CRAN (since 2015). The **parallelly** package implements enhancements to the **parallel** package originally part of the **future** package. The **digest** package (Eddelbuettel et al., 2021) is used to produce universally unique identifiers (UUIDs). Development is done toward a public Git repository hosted at `https://github.com/HenrikBengtsson/future`.

Validation

Since correctness and reproducibility is essential to all data processing, validation is a top priority and part of the design and implementation throughout the future ecosystem. Several types of testing are performed.

First, all the essential core packages part of the future framework, **future**, **parallelly**, **globals**, and **listenv**, implement a rich set of package tests. These are validated regularly across the wide range of operating systems (Linux, Solaris, macOS, and MS Windows) and R versions available on CRAN, on continuous integration (CI) services (GitHub Actions, Travis CI, and AppVeyor CI), and on R-hub.

Second, for each new release, these packages undergo full reverse-package dependency checks using **revdepcheck** (Csárdi and Wickham, 2021). As of November 2021, the **future** package is tested against 210 direct reverse-package dependencies available on CRAN and Bioconductor. These checks are performed on Linux with both the default settings and when forcing tests to use multisession workers (SOCK clusters), which further validates that globals and packages are identified correctly.

Third, a suite of *Future API conformance tests* available in the **future.tests** package (Bengtsson, 2021f) validates the correctness of all future backends. Any new future backend developed must pass these tests on complying with the *Future API*. By conforming to this API, the end-user can trust that the backend will produce the same correct and reproducible results as any other backend, including the ones that the developer has tested on. Also, by making it the responsibility of the backend developer to assert that their new future backend conforms to the *Future API*, we relieve other developers from having to test that their future-based software works on all backends. It would be a daunting task for a developer to validate the correctness of their software with all existing backends. Even if they would achieve that, there may be additional third-party future backends that they are not aware of, that they do not have the possibility to test with, or that yet have not been developed.

Fourth, since **foreach** is used by a large number of essential CRAN packages, it provides an excellent opportunity for supplementary validation. Specifically, we dynamically tweak the examples of **foreach** and popular CRAN packages **caret**, **glmnet**, **NMF**, **plyr**, and **TSP** to use the **doFuture** adaptor (Bengtsson, 2021a). This allows us to run these examples with a variety of future backends to validate that the examples produce no run-time errors, which indirectly validates the backends as well as the *Future API*. In the past, these types of tests helped to identify and resolve corner cases where automatic identification of global variables would fail. As a side note, several of these foreach-based examples fail when using a parallel foreach adaptor because they do not properly export globals or declare package dependencies. The exception is when using the sequential *doSEQ* adaptor (default), fork-based ones such as **doMC**, or the generic **doFuture**, which supports any future backend and relies on the future framework for handling globals and packages[13].

Lastly, analogously to the above reverse-dependency checks of each new release, CRAN and Bioconductor continuously run checks on all these direct, but also indirect, reverse dependencies, which further increases the validation of the *Future API* and the future ecosystem at large.

Known limitations

When saving an R object to file or sending it to a parallel worker, R uses a built-in technique called *serialization*, which allows a complex object structure to be sent as a stream of bytes to its destination, so it later can be reconstructed via *unserialization*. The ability to serialize objects is fundamental to all parallel processing, the exception being shared-memory strategies such as forked parallel processing. For example, this is how future expressions and variables are sent to parallel workers and how results are returned.

[13]There is a plan to update **foreach** to use the exact same static-code-analysis method as the **future** package use for identifying globals. As the maintainer of the future framework, I collaborate with the maintainer of the **foreach** package to implement this.

However, some types of objects are by design bound to the R session where they are created and cannot be used as-is in other R processes. One example is R *connection*s, e.g.,

```
con <- file("/path/to/file", open = "wb")
str(con)
#  'file' int 3
#  - attr(*, "conn_id")=<externalptr>
```

Any attempt to use a connection in another R process, for instance, by saving it to file, restarting R, and loading it back in, or by sending it to a parallel worker, will at best produce a run-time error, and in the worst case, produce invalid results or, for instance, write to the wrong file. These constraints apply to all types of parallelization frameworks in R, including the future framework.

There are other types of objects that cannot be transferred as-is to external processes, many from popular third-party packages, e.g., database connections of the **DBI** package, XML documents of the **xml2** package, STAN models of the **stan** package, and many more[14]. An indicator of this is when an R object has an *external pointer*, which is used for referencing an internal low-level object. This suggests that the object is bound to the current process and its lifespan. Unfortunately, it is not a sufficient indicator because some objects with external pointers can be exported, e.g., **data.table** objects. This makes it complicated to automate the detection of non-exportable objects and protect against using them in parallel processing. The current best practice is to be aware of these types of objects and to document new ones when discovered, which often happens when there is an unexpected run-time error. To help troubleshooting, it is possible to configure the **future** package to scan for and warn about globals with external pointers whenever used in a future.

Finally, it is theoretically possible to restructure some of the "non-exportable" object types such that they can be used in parallel processing. This is discussed further in the 'Future work' section.

Overhead

With parallel processing comes overhead. Typically, sources of added processing time are from spawning new parallel processes, sending instructions and globals to the workers, querying workers for results, and receiving results (Figure 1). Because of this, there is always a trade-off between sequential and parallel processing, and on how many parallel workers can be used before the total overhead dominates the benefits. Whether or not parallelization is beneficial, and for which parallel backends, depends on what is being parallelized.

As with other parallel solutions, in the future framework, overhead differs between parallel backends. Certain parallel backends, such as forked processing ("multicore"), are better suited for low-latency requirements, whereas others, such as distributed processing ("cluster" and "batchtools"), are better suited for large-throughput requirements. For example, many fast operations applied to a single large data frame should probably be parallelized on the local computer with forked processing, if supported, rather than being distributed on a compute cluster running in the cloud. In contrast, processing hundreds of data files may be completed sooner if distributed out to multiple computers (with access to the same file system), for instance, via a job scheduler, rather than being processed in parallel on the local machine.

Besides the overhead added by the parallel backend, each future, regardless of backend, has a baseline overhead. Specifically, there is a small overhead from the static-code inspection used to identify global variables, from exception handling needed to capture and relay errors, and from capturing and relaying standard output and conditions. Except for the error-handling overhead, these can all be avoided via certain `future()` arguments, e.g., by manually specifying globals needed and by disabling the relaying of output and conditions.

R has several profiling tools that can help identify bottlenecks and overhead in computational expensive tasks, e.g., `system.time()` of the **base** package, **microbenchmark** (Mersmann, 2021), **bench** (Hester, 2020), `Rprof()` of the **utils** package, **proffer** (Landau, 2021a), and **profvis** (Chang et al., 2020). These tools can also identify the different sources of overhead in the parallelization framework itself, including the ones in the future ecosystem. It is on the roadmap to make futures collect and report on some of these benchmarks automatically in order to help developers optimize their code and for end-users to choose a proper backend.

[14] See **future** package vignette '`Non-exportable object`' for more examples.

Results

The *Future API* is designed to unify parallel processing in R at the lowest possible level. It provides a standard for building richer, higher-level parallel frontends without having to worry about and reimplement common, critical tasks such as identifying global variables and packages, parallel RNG, and relaying of output and conditions - cumbersome tasks that are often essential to parallel processing.

Another advantage of the future framework is that new future backends do not have to implement their versions of these tasks, which not only lowers the threshold for implementing new backends, but also results in a consistent behavior throughout the future ecosystem, something none of the other parallel solutions provide. This benefits the developer because they can focus on what to parallelize rather than how and where. It also benefits the end-user, who will have more alternatives to how and where parallelization will take place. For instance, the developer might have local parallelization in mind during the development phase due to their work-environment constraints, whereas the end-user might be interested in parallelizing out to a cloud computing service. One may say that code using futures scales far without the developer's attention. Moreover, code using futures for parallelization will be able to take advantage of new backends that may be developed several years from now.

Directly related to the separation of code and backends, end-users and developers no longer need to rely on other package maintainers to update their code to take advantage of any new types of computational resources; updates that otherwise require adding another argument and conditional statement. One example of this was **future.batchtools**' predecessor, **future.BatchJobs** (legacy, CRAN, archived), which was straightforward to implement on top of **BatchJobs** (Bischl et al., 2015) as soon as the *Future API* was available. With zero modifications, code that previously only parallelized on the local computer could suddenly parallelize across thousands of cores on high-performance compute (HPC) clusters via the job scheduler. All it took was to change the `plan()`.

Because the future ecosystem is at its core designed to give consistent results across all sequential and parallel backends, it is straightforward to update, or port, an existing, sequential, map-reduce framework such that it can run in parallel. Not having to worry about low-level parallelization code, which otherwise risks blurring the objectives, lowers the threshold for designing and implementing new parallel map-reduce APIs. There are several examples of how fairly straightforward it is to implement higher-level parallel APIs on top of the *Future API*. The **future.apply** package (Bengtsson, 2021c), implements futurized variants of R's apply functions found in the **base** package, e.g., `future_apply()` and `future_lapply()` are plug-in replacements for `apply()` and `lapply()`. The **furrr** package (Vaughan and Dancho, 2021) implements futurized variants of the different map-reduce functions found in the **purrr** package (Henry and Wickham, 2020), e.g., `future_map()` is as plug-in replacement for `map()`. The **doFuture** package implements a generic **foreach** adaptor for `y <- foreach(...) %dopar% { ... }` that we can use with any future backend. Because the **BiocParallel** (Morgan et al., 2021) package, part of the Bioconductor Project, supports foreach as its backend, its functions such as `bplapply()` and `bpvec()` can also parallelize using *any type of future backend* via **doFuture**.

By lowering the barrier for implementing futurized variants of popular map-reduce APIs, developers and end-users are allowed to stay with their favorite coding style while still taking full advantage of the future framework.

The *Future API* also addresses the lock-in-versus-portability problem mentioned in the introduction; the risk that package developers on Unix-like systems would only support multicore parallelization methods because "`mclapply()` just works" is significantly lower using futures. Similarly, the most common way to parallelize code is to use multiple cores on the local machine. Because it is less common to have access to multiple machines, this often prevents developers from considering any other types of parallelization, with the risk of locking in end-users with other types of resources to only use a single machine. Hence, the chance for a package to support multi-host parallelization, including in the cloud and HPC environments, increases when using futures.

The burden on package developers to test and validate their parallel code is significant when using traditional parallelization frameworks, especially when attempting to support multiple variants. In contrast, when using futures, the cost of developing, testing, and maintaining parallel code is lower - often not much more than maintaining sequential code. This is possible because of the simplicity of the *Future API* and the fact that the orchestration of futures is predominantly done by the **future** package. Therefore, by implementing rigorous tests for the future framework and the different backend packages, the need for performing complementary tests in packages that make use of futures is much smaller. Tests for future backend packages, as well as the *Future API*, are provided by the **future.tests** package, which lowers the risk for a backend not being sufficiently tested.

The built-in protection against nested parallelism by mistake, and the agility of system settings of `availableCores()`, makes parallel code that uses futures to play nicely on multi-tenant systems. It respects all known R options and environment variables that specify, or otherwise limit the number

of parallel workers allowed. See `help("availableCores", package="parallelly")` for details. In contrast, it is, unfortunately, very common to find parallel code that uses `parallel::detectCores()` as the default number of workers in other parallel frameworks. Defaulting to using all available cores this way often wreak havoc on multi-tenant compute systems by overusing already consumed CPU resources, sometimes bringing the system to a halt due to too much context switching and memory use. Unfortunately, this often results in a negative performance on also other users' processes, and system administrators have to spend time tracking down the root cause of such poorly performing compute hosts.

Use of the future framework on CRAN and Bioconductor

The **future** package was released on CRAN in 2015. The uptake has grown steadily ever since. As of November 2021, **future** is among the top-1.1% most downloaded package on CRAN[15], and there are 210 packages on CRAN and Bioconductor that directly depend on it. For map-reduce parallelization packages **future.apply** (top-1.3% most downloaded) and **furrr** (top-1.8%), the corresponding number of packages are 87 and 58, respectively.

Besides supporting these traditional parallelization methods, the future framework is also used as an infrastructure elsewhere. For example, the workflow package **targets** (Landau, 2021b), and its predecessor **drake** (Landau, 2018), implements "a pipeline toolkit for reproducible computation at scale". They work by defining make-like targets and dependencies that can be resolved in parallel using any type of future backend. Another prominent example is the **shiny** package (Chang et al., 2021), which implements support for *asynchronous processing* in Shiny applications via futures. Asynchronous processing helps to avoid long-running tasks from blocking the user interface. Similarly, the **plumber** package (Schloerke and Allen, 2021), which automatically generates and serves HTTP API from R functions, uses futures to serve asynchronous web APIs and process tasks in parallel.

Other uses of futures

In Hewitt and Baker (1977), the authors propose the `(EITHER ...)` construct that "evaluates the expressions in parallel and return the value of 'the first one that finishes'." A corresponding R construct could be `future_either(...)` that evaluates R expressions concurrently via futures and returns the value of the first resolved one ignoring the others, e.g.,

```
y <- future_either(
  sort.int(x, method = "shell"),
  sort.int(x, method = "quick"),
  sort.int(x, method = "radix")
)
```

We may also use futures in cases that do not require parallel processing per se. Indeed, the *Future API* strives to make no assumptions about futures being resolved via parallel or distributed processing. One example is where a particular expression can only be resolved in a legacy version of R, on another operating system than where the main R session runs, or in an environment that meet specific requirements, e.g., large amounts of memory, fast local disks, or access to a certain genomic database. Another example of a resource-specific backend is the **civis** package (Miller and Ingersoll, 2020), which uses futures to provide an R client for the commercial Civis Platform.

We can also use futures to evaluate non-trustworthy R expressions in a sandboxed R environment that is, for instance, locked down in a virtual machine, or in a Linux container, such as Singularity (Kurtzer et al., 2017) or Docker (Merkel, 2014), without access to the host machine and its file system and network.

Future work

Although they are not part of the core future framework, future-based map-reduce packages **future.apply**, **furrr**, **doFuture**, and the like, play an essential role in how developers and end-users interact with futures. A key feature of these packages is "load balancing", which helps reduce the overall overhead that comes from setting up futures and spawning them on parallel workers and collecting their results. They achieve this by partitioning the elements to iterated over into equally sized chunks, typically so that there is one chunk per worker, which in turn results in one future per

[15]The ranks are robust estimates based on the average median weekly download counts from the RStudio CRAN mirror during four weeks.

chunk and hence one future per worker. In contrast, without load balancing, each element is processed by one future resulting in more overhead, especially when there are many elements to iterate over. Each of these packages has its own implementation of load balancing, despite often using exactly the same algorithm. If there is an improvement or a bug fix to one, the maintainers of the others need to update their code too. The same is true for how they orchestrate globals and parallel RNG. To improve on this situation and to further harmonize the behavior of futures in these packages, a new helper package **future.mapreduce** that implements these common tasks will be introduced, relieving these packages from those tasks. This will also have the advantage of making it even easier to implement other types of map-reduce APIs on top of futures.

Having said this, in a longer perspective, it might be possible to remove the need for these future-based map-reduce APIs, which essentially are thin wrappers ported from their counterpart map-reduce APIs. This would require internal refactoring of the core future framework, but it can likely be done while preserving full backward compatibility with the current *Future API*. For clarification, consider the following `lapply()` construct that evaluates `slow_fcn(x)` for ten elements, each resolved via a unique *lazy* future:

```
xs <- 1:10
fs <- lapply(xs, function(x) future({
  slow_fcn(x)
}, lazy = TRUE))
```

A lazy future defers the evaluation of its expression until we use `resolved()` to query if it is resolved or until we use `value()` to collect its value[16]. Since neither has been called above, these futures are still dormant, regardless of future backend used. Next, assume that there are two parallel workers and imagine that we have a function `merge()` to merge futures. This would allow us to partition ten futures into only two futures, one per worker, and then collect their values:

```
f1 <- merge(fs[1:5])
f2 <- merge(fs[6:10])
vs <- c(value(f1), value(f2))
```

We can simplify this further by encapsulating the above in the S3 method `value()` for *list*s:

```
vs <- value(fs)
```

We can mitigate the verbosity in the setup of futures with a helper function or syntax sugar. More importantly, this would make it possible to use futures in map-reduce APIs without the need for a counterpart parallel implementation. It would also lower the threshold further for adding a thin layer of support for futures *within* existing map-reduce APIs, especially since the design of the future framework keeps the added maintenance burden to a minimum.

A frequently requested feature is to support *suspending* running futures, particularly when their runtimes are large. For example, above `future_either()` function could benefit from a `suspend()` function to terminate futures no longer needed. Since not all backends may support this, extra care needs to be taken when introducing this feature to the future framework. A related feature request is the possibility to *restart* a future that failed due to, for instance, a crashed worker or a partial power failure on a compute cluster, e.g., `restart(f)`. Combined with R's condition handling framework, higher-level APIs can then take on the role of retrying to resolve failed futures, e.g., `retry({ ... }, times = 3, on = "FutureError")`.

Implementing support for suspending and restarting futures will indirectly add support for serializing futures themselves, which is only partially supported in the current implementation. Being able to serialize futures opens up further possibilities such as saving futures to be processed at a later time, in another context, or transferring them to a job queue that, in turn, distributes them to appropriate compute resources.

The problem of not being able to export all types of objects as-is in parallel processing can be a blocker. It turns out that for a subset of these, we could use *marshaling* to encode them before serializing them such that a working clone can be reconstructed after unserializing and *unmarshaling*. As an example, a read-only file connection can be marshaled by recording its filename and file position so that the parallel worker could open its own read-only connection for the same file at the same position. Marshaling is a rarely used concept in R, possibly because there is no standard convention for package developers to rely on. Ideally, such a mechanism would allow package developers to register custom `marshal()` and `unmarshal()` methods for their data types, making them automatically applicable in parallelization without prior knowledge of what objects being transferred.

[16] Although a lazy future defers the evaluation to a later time, contrary to R's *lazy evaluation* and *promises*, a future records all dependent variables ("globals") when it is created, which means it will resolve to the same value even if those globals change after the future was created and before it was resolved. This also means that lazy and eager futures give the same value.

Other than setting the backend via `plan()`, it is not possible to direct a particular future to a specific backend type based on the needs of the future. To support this, we have to add options to declare what *resources* are needed to resolve particular future. For instance,

```
f <- future({ ... }, resources = c("r:3.2.*", "mount:/data", "!fork"))
```

could be one way to specify that this future has to be resolved with R 3.2 on a machine with a `/data` mount point and that forked parallel processing must not be used. Some resources may be implicit based on exported globals, e.g., a specific file required when exporting a file connection via marshaling.

All the above is on the roadmap for the future framework.

Summary

The **future** package is a lightweight R package that provides an alternative approach for parallel processing in R. It implements the *Future API*, which comprises three basic functions, from which richer, higher-level APIs for parallel processing can be constructed. Several of these higher-level APIs mimic counterpart map-reduce APIs closely, allowing developers to stay with their favorite coding style for their parallel needs. The future framework is designed so that the developer does not have to worry about common, critical tasks such as exporting globals to workers, using proper parallel RNG, and taking care of output, messages, warnings, and errors. This design lowers the barriers to reimplement existing algorithms and methods in parallel while avoiding increasing the maintenance burden. When using futures, the end-user controls which parallel backend is used, while the developer controls what to parallelize. This is possible because all future backends have been validated to conform to the *Future API* specifications, ensuring that futures produce the same results regardless of how and where they are processed.

Acknowledgments

I am grateful to all users and developers who have contributed to the future framework with questions and answers, feature requests, bug reports, and interesting and fruitful discussions. I am thankful to the reviewers, the editor, and everyone else who gave comments and suggestions helping to improve this article. The development of the *Future API Specifications and Conformance* test suite in the **future.tests** package was supported by the R Consortium through its Infrastructure Steering Committee (ISC) grant program.

Bibliography

H. Bengtsson. *listenv: Environments Behaving (Almost) as Lists*, 2019. URL https://CRAN.R-project.org/package=listenv. R package version 0.8.0. [p282]

H. Bengtsson. *globals: Identify Global Objects in R Expressions*, 2020. URL https://CRAN.R-project.org/package=globals. R package version 0.14.0. [p281]

H. Bengtsson. *doFuture: A Universal Foreach Parallel Adapter using the Future API of the 'future' Package*, 2021a. URL https://CRAN.R-project.org/package=doFuture. R package version 0.12.0. [p284]

H. Bengtsson. *future: Unified Parallel and Distributed Processing in R for Everyone*, 2021b. URL https://CRAN.R-project.org/package=future. R package version 1.23.0. [p273]

H. Bengtsson. *future.apply: Apply Function to Elements in Parallel using Futures*, 2021c. URL https://CRAN.R-project.org/package=future.apply. R package version 1.8.1. [p286]

H. Bengtsson. *future.batchtools: A Future API for Parallel and Distributed Processing using 'batchtools'*, 2021d. URL https://CRAN.R-project.org/package=future.batchtools. R package version 0.10.0. [p283]

H. Bengtsson. *future.callr: A Future API for Parallel Processing using 'callr'*, 2021e. URL https://CRAN.R-project.org/package=future.callr. R package version 0.6.1. [p283]

H. Bengtsson. *future.tests: Test Suite for Future API Backends*, 2021f. URL https://CRAN.R-project.org/package=future.tests. R package version 0.3.0. [p284]

H. Bengtsson. *parallelly: Enhancing the 'parallel' Package*, 2021g. URL https://CRAN.R-project.org/package=parallelly. R package version 1.28.1. [p284]

H. Bengtsson. *progressr: A Inclusive, Unifying API for Progress Updates*, 2021h. URL https://CRAN.R-project.org/package=progressr. R package version 0.9.0. [p280]

B. Bischl, M. Lang, O. Mersmann, J. Rahnenführer, and C. Weihs. BatchJobs and BatchExperiments: Abstraction mechanisms for using R in batch environments. *Journal of Statistical Software*, 64(11): 1–25, 2015. URL https://dx.doi.org/10.18637/jss.v064.i11. [p286]

W. Chang, J. Luraschi, and T. Mastny. *profvis: Interactive Visualizations for Profiling R Code*, 2020. URL https://CRAN.R-project.org/package=profvis. R package version 0.3.7. [p285]

W. Chang, J. Cheng, J. Allaire, Y. Xie, and J. McPherson. *shiny: Web Application Framework for R*, 2021. URL https://CRAN.R-project.org/package=shiny. R package version 1.7.1. [p287]

J. Cheng. *promises: Abstractions for Promise-Based Asynchronous Programming*, 2021. URL https://CRAN.R-project.org/package=promises. R package version 1.2.0.1. [p281]

G. Csárdi and W. Chang. *callr: Call R from R*, 2021. URL https://CRAN.R-project.org/package=callr. R package version 3.7.0. [p283]

G. Csárdi and H. Wickham. *revdepcheck: Automated Reverse Dependency Checking*, 2021. URL https://github.com/r-lib/revdepcheck#readme. R package version 1.0.0.9001. [p284]

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004. URL https://doi.org/10.1145/1327452.1327492. [p273]

D. Eddelbuettel. Parallel computing with R: A brief review. *WIREs Computational Statistics*, 13(2):e1515, 2021. doi: 10.1002/wics.1515. URL https://doi.org/10.1002/wics.1515. [p273]

D. Eddelbuettel, with contributions by Antoine Lucas, J. Tuszynski, H. Bengtsson, S. Urbanek, M. Frasca, B. Lewis, M. Stokely, H. Muehleisen, D. Murdoch, J. Hester, W. Wu, Q. Kou, T. Onkelinx, M. Lang, V. Simko, K. Hornik, R. Neal, K. Bell, M. de Queljoe, I. Suruceanu, and B. Denney. *digest: Create Compact Hash Digests of R Objects*, 2021. URL https://CRAN.R-project.org/package=digest. R package version 0.6.28. [p284]

M. Edmondson. *googleComputeEngineR: R Interface with Google Compute Engine*, 2019. URL https://CRAN.R-project.org/package=googleComputeEngineR. R package version 0.3.0. [p283]

D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, apr 1978. doi: 10.1109/tc.1978.1675100. URL https://doi.org/10.1109/tc.1978.1675100. [p273]

L. Henry and H. Wickham. *purrr: Functional Programming Tools*, 2020. URL https://CRAN.R-project.org/package=purrr. R package version 0.3.4. [p286]

J. Hester. *bench: High Precision Timing of R Expressions*, 2020. URL https://CRAN.R-project.org/package=bench. R package version 1.1.1. [p285]

C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977. URL https://dblp.uni-trier.de/db/conf/ifip/ifip1977.html#HewittB77. [p273, 287]

P. Hibbard. Parallel processing facilities. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*. IRIA, 1976. [p273]

M. Kane, J. Emerson, and S. Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013. ISSN 1548-7660. doi: 10.18637/jss.v055.i14. URL https://doi.org/10.18637/jss.v055.i14. [p275]

G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS One*, 12(5):e0177459, 2017. URL https://doi.org/10.1371/journal.pone.0177459. [p287]

W. M. Landau. The drake R package: A pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 3(21), 2018. URL https://doi.org/10.21105/joss.00550. [p287]

W. M. Landau. *proffer: Profile R Code and Visualize with 'Pprof'*, 2021a. URL https://CRAN.R-project.org/package=proffer. R package version 0.1.5. [p285]

W. M. Landau. The targets R package: a dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 6(57):2959, 2021b. URL https://doi.org/10.21105/joss.02959. [p287]

M. Lang, B. Bischl, and D. Surmann. batchtools: Tools for R to work on batch systems. *Journal of Open Source Software*, 2(10):135, feb 2017. doi: 10.21105/joss.00135. URL https://doi.org/10.21105/joss.00135. [p283]

P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999. URL https://doi.org/10.1287/opre.47.1.159. [p281]

B. W. Lewis. *doRedis: 'Foreach' Parallel Adapter Using the 'Redis' Database*, 2020. URL https://CRAN.R-project.org/package=doRedis. R package version 3.0.0. [p275]

J. Luraschi, K. Kuo, K. Ushey, J. Allaire, H. Falaki, L. Wang, A. Zhang, Y. Li, and The Apache Software Foundation. *sparklyr: R Interface to Apache Spark*, 2021. URL https://CRAN.R-project.org/package=sparklyr. R package version 1.7.2. [p273]

M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, pages 3–30, 1998. URL https://doi.org/10.1145/272991.272995. [p281]

D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014. [p287]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2021. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-8. [p285]

Microsoft and S. Weston. *foreach: Provides Foreach Looping Construct*, 2020. URL https://CRAN.R-project.org/package=foreach. R package version 1.5.1. [p275]

Microsoft Corporation and S. Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*, 2020. URL https://CRAN.R-project.org/package=doParallel. R package version 1.0.16. [p275]

P. Miller and K. Ingersoll. *civis: R Client for the 'Civis Platform API'*, 2020. URL https://CRAN.R-project.org/package=civis. R package version 3.0.0. [p287]

M. Morgan, V. Obenchain, M. Lang, R. Thompson, and N. Turaga. *BiocParallel: Bioconductor Facilities for Parallel Evaluation*, 2021. URL https://bioconductor.org/packages/BiocParallel/. R package version 1.28.0. [p286]

Revolution Analytics and S. Weston. *doMC: Foreach Parallel Adaptor for 'parallel'*, 2020. URL https://CRAN.R-project.org/package=doMC. R package version 1.3.7. [p275]

B. Schloerke and J. Allen. *plumber: An API Generator for R*, 2021. URL https://CRAN.R-project.org/package=plumber. R package version 1.1.0. [p287]

M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State-of-the-art in parallel computing with r. *Journal of Statistical Software*, 47(1), 2009. [p273]

D. Schmidt, W.-C. Chen, M. A. Matheson, and G. Ostrouchov. Programming with BIG data in R: Scaling analytics from one to thousands of nodes. *Big Data Research*, 8:1–11, 2017. doi: https://doi.org/10.1016/j.bdr.2016.10.002. [p273]

L. Tierney. *codetools: Code Analysis Tools for R*, 2020. URL https://CRAN.R-project.org/package=codetools. R package version 0.2-18. [p281]

L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2021. URL https://CRAN.R-project.org/package=snow. R package version 0.4-4. [p274]

S. Urbanek. *multicore: A Stub Package to Ease Transition to 'parallel'*, 2014. URL https://CRAN.R-project.org/package=multicore. R package version 0.2. [p274]

D. Vaughan and M. Dancho. *furrr: Apply Mapping Functions in Parallel using Futures*, 2021. URL https://CRAN.R-project.org/package=furrr. R package version 0.2.3. [p286]

S. Weston. *doMPI: Foreach Parallel Adaptor for Rmpi Package*, 2017. URL https://CRAN.R-project.org/package=doMPI. R package version 0.2.2. [p275]

Henrik Bengtsson
Department of Epidemiology and Biostatistics,
University of California, San Francisco
San Francisco, CA
United States
henrik.bengtsson@gmail.com