Parallel execution using ParallelLogger

Martijn J. Schuemie

2025-08-27

Contents

1 Introduction

This vignette describes how you can use the `ParallelLogger` package to execute R code in parallel. This can help speed up an analysis, by using multiple processors at the same time instead of using a single processor to execute the code in sequence.

1.1 Important concepts.

- **Cluster**: A set of compute nodes.
- **Node**: A separate instance of R that is instantiated, controlled, and terminated by the user's current R session.

In this package, all nodes are on the same computer. Note that the `parallel` package allows you to also create nodes on remote machines, for example in a physical computer cluster.

2 Creating a cluster

We can create a cluster using the `makeCluster` command:

```
cluster <- makeCluster(numberOfThreads = 3)
```

```
## Initiating cluster with 3 threads
```

This instantiates three new R instances on your computer, which together form a cluster.

2.1 Single-node cluster

Note that if we set `numberOfThreads = 1`, the default behavior is to not instantiate any nodes. Rather, the main thread (the user's session) is used for execution. One advantage of this is that it is easier to debug code in the main thread, as it is possible for example to set break points or tag a function using `debug`, which is not possible when using multiple threads. To disable this behavior, you can set `singleThreadToMain = FALSE`.

3 Executing in parallel

Any code that we want to execute needs to be implemented in an R function, for example:

```
fun <- function(x, constant) {
     return(x * constant)
}
```

This simple function merely computes the product of `x` and `constant`, and returns it.

We can now execute this function in parallel across our cluster:

```
x <- 1:3
clusterApply(cluster, x, fun, constant = 2)
```

```
##   |                                                                      |
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
```

The function `clusterApply` executes the function across the nodes over all values of x, and returns all responses in a list. Note that by default a progress bar is shown. This can be disabled by setting `progressBar = FALSE` when calling `clusterApply`.

Important: `clusterApply` does not guarantee that the results are returned in the same sequence as `x`.

Important: The context in which a function is created is also transmitted to the worker node. If a function is defined inside another function, and that outer function is called with a large argument, that argument will be transmitted to the worker node each time the function is executed, causing substantial (probably unnecessary) overhead. It can therefore make sense to define the function to be called at the package level rather than inside a function, to save overhead. So for example, in the code below every time the function `doTinyJob` is called, the `largeVector` argument is passed to the worker node:

```
doBigJob <- function(largeVector) {
  doTinyJob <- function(x){
    return(x^2)
  }
  cluster <- makeCluster(numberOfThreads = 3)
  clusterApply(cluster, largeVector, doTinyJob)
  stopCluster(cluster)
}
```

It is much more efficient to declare `doTinyJob` outside the `doBigJob` function:

```r
doTinyJob <- function(x){
  return(x^2)
}

doBigJob <- function(largeVector) {
  cluster <- makeCluster(numberOfThreads = 3)
  clusterApply(cluster, largeVector, doTinyJob)
  stopCluster(cluster)
}
```

4 Stopping the cluster

Once you are done using the cluster, make sure to stop it:

```r
stopCluster(cluster)
```

```
## Stopping cluster
```

5 Debugging, logging, and error handling

As mentioned in the Single-node cluster section, R's standard debugging tools do not work when executing in parallel. Setting `numberOfThreads = 1` when calling `makeCluster` ensures the code is executed in the main thread, so breakpoints and debugging function properly.

The *Logging using ParallelLogger* vignette explains how logging can be used to record events, including warnings and errors, when executing in parallel.

By default, an error thrown by a node does not cause the execution in the other nodes to stop. Instead, when an error is thrown the execution continues over the other values, and not until those are complete is an error thrown in the main thread. The rationale for this is that it might be informative to see all errors instead of just the first. However, this behavior can be changed by setting `stopOnError = TRUE` when calling `clusterApply`.

6 Support for Andromeda objects

The `Andromeda` package allows the user to work with data objects that are too large to fit in memory, by storing the data on disk. When calling `makeCluster`, the `andromedaTempFolder` option is copied from the main thread to all worker nodes. Note that it is not possible to pass Andromeda objects to nodes. A workaround could be to pass the file name of an Andromeda object instead.

As of version 1.0.0, `Andromeda` recognizes the `andromedaMemoryLimit` option. The default behavior of `ParallelLogger` is to set the `andromedaMemoryLimit` in each thread to be either the global `andromedaMemoryLimit / numberOfThreads` (if the global option is set) or 75% of the system memory / number of threads.