Received: 19 July 2017 | Revised: 15 March 2019 | Accepted: 8 April 2019

DOI: 10.1002/sim.8188

TUTORIAL IN BIOSTATISTICS

WILEY Statistics in Medicine

Cloud-based simulation studies in R - A tutorial on using doRedis with Amazon spot fleets

G. Hirschfeld | C. Thiele

University of Applied Sciences Bielefeld, Bielefeld, Germany

Correspondence
G. Hirschfeld, University of Applied Sciences Bielefeld, 33619 Bielefeld, Germany.
Email: g.hirschfeld@fh-bielefeld.de

Funding information
Bundesministerium für Bildung und Forschung, Grant/Award Number: 01EK1501

Simulation studies are helpful in testing novel statistical methods. From a computational perspective, they constitute embarrassingly parallel tasks. We describe parallelization techniques in the programming language R that can be used on Amazon's cloud-based infrastructure. After a short conceptual overview of the parallelization techniques in R, we provide a hands-on tutorial on how the doRedis package in conjunction with the Redis server can be used on Amazon Web Services, specifically running spot fleets. The tutorial proceeds in seven steps, ie, (1) starting up an EC2 instance, (2) installing a Redis server, (3) using doRedis with a local worker, (4) using doRedis with a remote worker, (5) setting up instances that automatically fetch tasks from a specific master, (6) using spot-fleets, and (7) shutting down the instances. As a basic example, we show how these techniques can be used to assess the effects of heteroscedasticity on the equal-variance t-test. Furthermore, we address several advanced issues, such as multiple conditions, cost-management, and chunking.

KEYWORDS

cloud computing, parallel computing, R, Redis, simulation study

1 | INTRODUCTION

Simulation studies play an increasinlgy important role in medical statistics. First, they are important tools for assessing novel statisical methods and their limitations in known scenarios.[1-3] Even in situations where closed-form derivations are possible, simulation studies may be more accessible to applied researchers.[4] Second, simulations are effective teaching tools in statistics education.[5] Within the last decade, the open-source programming language **R**[6] has become the de facto standard for the development of novel statistical methods in medicine. It is vital that simulation studies can be performed efficiently in this language. While base **R** already encompasses all of the necessary features to perform simulation studies, ie, generating random data and loops,[7] these often still lead to inefficient execution of simulation studies in that they by default only use one processing unit of the machine on which the code was started. Running more complex simulation studies comprising a large number of different scenarios, or many repetitions, can quickly become extremely time consuming.

However, since simulation studies usually entail repeating a specific analysis several thousand times on independent data sets, they are easy to parallelize, or what some call embarrasingly parallel tasks.[8] Due to the importance of parallelization techniques, several previous papers have already provided overviews of parallelization in **R**[9] and hands-on tutorials describing how to run code in parallel.[10,11] However, previous tutorials have focused mostly on local parallelization, ie, parallelizing code, so that all CPU cores on one machine, or local high-performance clusters, are used. Secondly, they have not always addressed some important issues for simulation studies in medical statistics, eg, setting random number seeds for reproducibility.[3]

The aim of the present tutorial is to enable researchers to set up and use cloud-based parallelization methods using Amazon Web Services to run simulation studies that are typical for medical statistics. We hope to enable researchers without access to high-performance infrastructure to run demanding simulation studies. The tutorial is organized as follows. First, we will give a conceptual overview of parallelization packages for **R** and cloud computing using Amazon Web Services. Then, we present a step-by-step tutorial that describes how to set up remote computing instances. Thereafter, we describe an example simulation study contrasting the effects of heteroscedasticity on the equal-variance t-test[12] and the Welch test.[13] Finally, we describe several advanced features of simulation studies, such as running complex simulation studies with several conditions, cost-management, and chunking.

1.1 | Simulation studies in R

Simulation studies in medicine often entail drawing thousands of independent samples and performing an analysis in each of those.[3] For example, a simulation study might want to test the empirical alpha-error rates of the equal-variance t-test. In a minimal example, a researcher repeatedly draws two samples from a normal distribution with known and equal means and standard deviations, performs a t-test to compare the groups, and records the resulting p-value. The following code* does just that. Specifically, the `for()` command is used to define a loop with 1000 iterations. In each iteration, two groups, ie, a control group and an experimental group, are sampled from a normal distribution with a mean of 50 and a standard deviation of 1. Then, the `t.test()` command is used to perform a classic t-test assuming equal variances and the resulting p-value is recorded. The additional call to `system.time()` is only necessary to determine how long it takes to perform this analysis.

```
> p_vals <- rep(NA, 1000)
> system.time(
+     for(i in 1:1000){
+         control <- rnorm(300000, mean = 50, sd = 1)
+         experimental <- rnorm(300000, mean = 50, sd = 1)
+         p_vals[i] <- t.test(control, experimental, var.equal = TRUE)$p.value
+     }
+ )

##    user system elapsed
##  82.540 0.592 83.163

> table(p_vals<.05)

##
## FALSE TRUE
##   939   61
```

About 5% of the t-tests yielded p-values smaller than .05. Since both samples are drawn from the same distribution, this is exactly what we expect if the test works as advertised. All in all, it took approximately 83 seconds to generate the independent samples, perform the t-test, and return the p-values.

1.2 | Parallelization in R

Since these 1000 samples are analyzed independently, it is possible that several CPUs work in parallel on the 1000 repetitions. However, the `for()` loop, which we just used, only uses one thread and works on the different repetitions sequentially. Luckily, several packages have been developed to address this shortcoming by providing functions to parallelize such embarassingly parallel tasks, eg, `foreach` or alternative constructs (for an overview see the work of Schmidberger et al[9]). A general overview of packages for parallelization in **R** is beyond the scope of this tutorial. These packages differ in their availability for different operating systems, the technical details of their parallelization

*Note that we decided to print prompt characters for the different code-blocks, ie, > for R-code and $ for bash-code. These have to be removed when copying and pasting code.

functionalities, and their support for reproducible random number generation. A convenient alternative is the `mclapply()` function from the **parallel** package that replaces the sequential `lapply()` function. However, it does not support parallelization on Windows. To run parallelized loops on such clusters `foreach` from the **foreach** package is a popular replacement for a standard `for()` loop. In order to use it, a package is needed that distributes the parallel tasks to a set of workers. These workers can either be physical computers or simply different cores on a single machine. The **doSNOW** package is able to start up local "clusters" and thus use parallelization on a single computer. It does not, however, offer more advanced features, such as fault tolerance and connection management, which are needed for running massively parallel jobs on a large number of machines. The latter can be achieved using **doRedis** as described later. Nevertheless, since most of the code for parallelization is identical using either **doSNOW** or **doRedis**, even larger simulation studies that ultimately run using **doRedis** in the cloud can be tested using **doSNOW** on a local machine.

1.2.1 | Parallelization using doSNOW

The following code gives an alternative using the **doSNOW** package[14] in combination with `foreach()`. The first part of the code loads the **doSNOW** package, and initiates and registers a cluster, ie, a group of processing units. Here, we decided to use all four available CPU threads of the local machine. Once this cluster is established, we can use `foreach()` in conjunction with `%dopar%` to cycle through a loop in parallel. Specifically, the foreach() function takes two parameters. The first, `i`, is used to set the number of repetitions, here 1000. The second, `.combine`, determines how the results are being combined. In this case, we want to gather the results in a vector. If the individual repetitions produce more complex output, eg, a vector or data frame, it is usually coerced into a table using `rbind` or `cbind`. The command `%dopar%`, as opposed to `%do%`, is needed for R to distribute the different parts of the loop to different workers as specified in the first lines. The body of the `foreach()` loop, ie, the commands within the curly brackets, define the tasks. Behind the scenes, the 1000 tasks will be distributed across the four threads and all four threads will work in parallel. To determine the number of threads that are available on the machine, the function `detectCores()` from the **parallel** package may be used. More information can be found in the vignette of the **foreach** package.[15]

```
> # Part 1: Load packages and set up the cluster
> library(doSNOW)
> cluster<-makeCluster(4, type = "SOCK")
> registerDoSNOW(cluster)
>
> # Part 2: Cycle though the foreach loop in parallel
> system.time(
+     p_vals <- foreach(i = 1:1000, .combine = c) %dopar% {
+           control <- rnorm(300000, 50, 1)
+           experimental <- rnorm(300000, 50, 1)
+           t.test(control, experimental, var.equal = TRUE)$p.value
+      }
+ )

##       user system elapsed
##      0.729 0.057 24.733

> stopCluster(cluster)
>
> table(p_vals<0.05)

##
## FALSE TRUE
## 940 60
```

Now, the time it takes to cycle through the different repetitions is only approximately 25 seconds. This near fourfold speedup is the maximum that can be expected from using four times as many threads. In practice, the speedup will usually be less than this optimum because the distribution of tasks and data is an operation that introduces an overhead.

1.2.2 | Parallelization using doRedis

The idea of distributing tasks across different CPUs can be expanded even further by not only including processing cores of the local machine but also remote computing nodes. One way of achieving this is by making use of the Redis server, **doRedis**, and the corresponding **rredis** package. The Redis server acts as the distributor of tasks to worker nodes (see Figure 1 for a diagram of the interdependence in the setup). Whenever a worker finishes a specific task, the results are returned to the server and combined there. Redis server is an open-source application that implements a networked in-memory key-value database. It supports strings, hashes, lists, sets, and sorted sets. Its throughput and latency are comparable to well-known database systems such as Cassandra and MySQL.[16] Accordingly, it is suited for holding, sending, and receiving data for statistical tasks that are exchanged between a master and a potentially high number of workers. For **doRedis**, the smallest unit of work is a "task", which represents one or more iterations of the body of a **foreach** loop. All tasks that are initiated by a single invocation of foreach are bundled into a "job", which is submitted to a "work queue". Internally, the latter is a "list" on the Redis server. This setup supports a dynamically growing or shrinking pool of workers, so that instances can be added or shut down at any time. As the submission and reception of tasks is tracked, tasks can be resubmitted. That way, unexpected shutdowns or failures of worker instances can be handled. Workers can listen to a queue without any jobs and jobs can be issued to a queue without workers. Workers will pick up any jobs that are submitted to the specified queue.[17] Something that **R**-users, who have worked with alternative packages for parallelization, have to get accustomed to is installing and then starting the Redis server from outside the **R** session.

For **doRedis**, it does not make a difference whether the master and the workers are started on the same machine or remotely. One could, for example, use a local master to which several remote workers connect, or even use the same machine as the master and worker. The setup that we will introduce in the tutorial consists of one remote master running a Redis server as well as an Rstudio Server instance and multiple remote workers that connect to this master. We will use Amazon's cloud computing infrastructure for the master and workers.

1.3 | Cloud computing and instance types on Amazon Web Services

Cloud computing means that, rather than using the local computer to perform an analysis, a number of remote computing instances are used. The main advantages for simulation studies are scalability and that only the computing power that is needed will be paid for. Elastic Compute Cloud (EC2) is Amazon's cloud computing service that offers a variety of so-called instance types from small ones with only one or two cores to instances with up to 96 cores. Currently (April 2018), the largest instance can be rented for about $4.60 an hour. This price can be reduced substantially by using "spot instances" that become an option for simulation studies in **R**, thanks to the fault tolerance of **doRedis**.

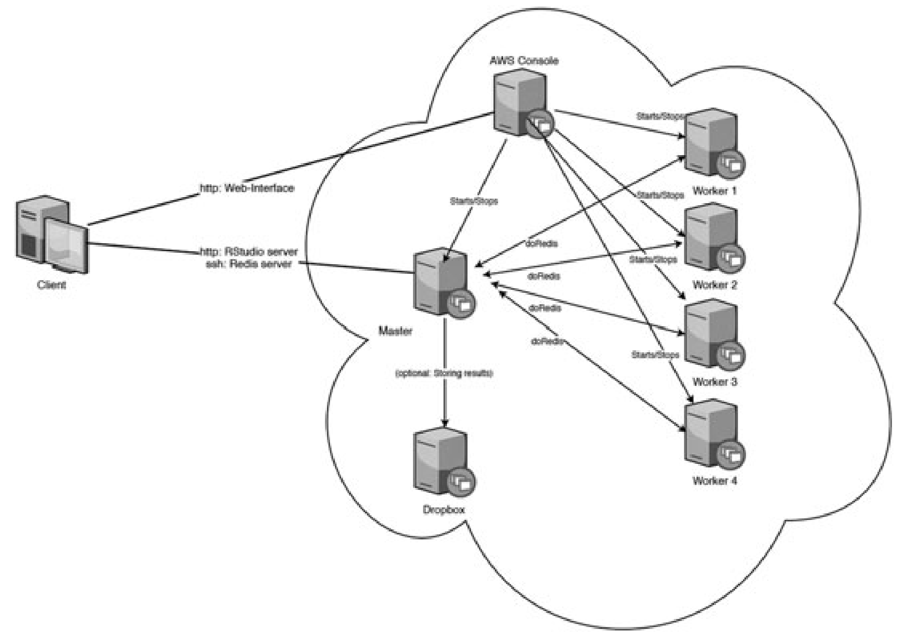

FIGURE 1 Setup and interplay of client, master, and workers using doRedis, Redis server, and RStudio server

Amazon offers two ways of renting an instance: "on demand" or "spot". "On demand" instances would not be shut down and have a fixed price that is usually above the price for spot instances. "Spot" instances have a dynamic price that depends on supply and demand and may be automatically shut down due to demand by Amazon. The user makes a bid, ie, the maximum amount per instance hour that they are willing to pay, and if this bid exceeds the current spot price, the requested instance(s) will start up and run as long as the spot price does not exceed the bid or the instances are shut down either by the user or automatically. The spot instance price is volatile and on average, approximately 60% less than the on-demand price.[18] Take the c4.4xlarge instance as an example. In the US East region, the current (April 2018) on-demand price is \$0.80, whereas the spot price is around \$0.14. However, spot prices fluctuate and may spike, exceeding the bid price.

Ultimately, we will use an on-demand instance for the master and one or several spot instances for the workers. The former runs RStudio, the Redis server, monitors the workers, and collects the results. Since this does not require much computing power, a smaller instance will suffice (for example t2.large). It is usually irrelevant for compute-intensive tasks which exact type of instance the workers are run on. The main consideration is how many CPUs are desired and a variety of different instances can be added to the cluster until that target capacity is reached.

2 | TUTORIAL

The tutorial is organized in seven steps, ie, (1) starting up an EC2 instance, (2) installing the Redis server, (3) using **doRedis** with a local worker, (4) using **doRedis** with a remote worker, (5) automatically fetch tasks from a specific master, (6) using spot-fleets, and (7) shutting down the instances.

2.1 | Step 1: starting up an EC2 instance and connect via Rstudio and SSH

2.1.1 | Starting the instance

In order to log in to the Amazon Web Services (AWS) console at aws.amazon.com, an Amazon account is needed. Next, we have to select "Amazon EC2" in the products menu. On clicking "Launch instance" or to the left in the "instances" menu, several different Amazon Machine Images (AMIs) can be selected. These include the operating system to be run on the instance(s) as well as any other software that comes with the AMI. For the purposes of this tutorial, we will use and further configure a community AMI, which is an AMI that was supplied by a volunteer. The AMI RStudio-1.0.143_R-3.4.0_Julia-0.5.2_ubuntu-16.04-LTS-64bit (for example, available as ami-4421f92b in the region EU (Frankfurt) and under different identifiers in other regions) runs Ubuntu 16.04 and already includes the RStudio server, which can be conveniently accessed using a browser after starting the instance. The final and suitably configured AMI that was used for running the below simulation is publicly available as the AMI ami-0af064a0333ed46f2 in the region EU (Frankfurt). After selecting the AMI, AWS displays the dialog for picking an instance type. Since we are only setting up our personal AMI, we can select a cheap or even free instance like t2.micro. The instance details on the next page and the storage size can be left as is. We also do not have to add tags. However, a security group has to be created that makes the instance and RStudio available on the web. We create a security group that opens the ports 22 for SSH, 80 for RStudio, and 6379 for Redis. For security reasons, the source IP can be set to the user's IP address. Otherwise, the instance will be open to the world if the default of `0.0.0.0` is kept. For testing purposes and if security is not a concern, the user can proceed using the default. The following page summarizes the chosen options. Upon "Launch", AWS will prompt the user for an SSH key. If no such key file was created, yet, AWS will create a key pair, which is necessary to gain SSH access to the instance.

2.1.2 | Access the RStudio server using a browser

The new instance can now be found under the "Instances" menu. By using a web browser to navigate to the instance's public DNS, we can log in to the RStudio server. The username and password should be taken from the description of the AMI (both are "rstudio" in older versions). We follow the instructions in the "Welcome.R" file to change the default password and to link our Dropbox account to the instance. That way, any .RData files or other files that are generated by our simulations and stored in the Dropbox folder can be synchronized and do not have to be downloaded manually.

2.1.3 | Access the instance via SSH

In addition to logging in to the machine using RStudio, it will be necessary to connect to the instance via SSH. To do so, we need the previously created `.pem` keyfile. The following instructions apply for Linux and Mac OS. For Windows, the instructions for connecting to an AWS instance using PuTTY should be followed.[†] If this file has not been used before, we have to change its permissions by entering

```
$ chmod 400 /path/my-key-pair.pem.
```

Then, we can access the instance as user "ubuntu" by entering

```
$ ssh -i /path/my-key-pair.pem ubuntu@[PUBLIC DNS],
```

where the path to the key file and the public DNS are replaced by the user's actual values, eg, `ec2-12-34-567-890.eu-central-1.compute.amazonaws.com`. If the login was successful, the usual terminal functions can be used, eg, updating the instance, installing other software, moving and deleting files, and so on. Most interestingly, we can use

```
$ top
```

to check the CPU load, if we are accessing a worker, and the Redis command line interface via

```
$ redis-cli
```

to gain access to the Redis server on the master as soon as it is installed. Later versions of RStudio and RStudio Server provide a Terminal tab that allows for executing commands directly in the RStudio session. For this application, this is useful for all commands that do not require super user privileges. Otherwise, authentication via the .pem file is necessary.

2.2 | Step 2: installing the Redis server on the Amazon instance

To install the Redis server, we first log in to the machine via SSH and then enter the following[‡]:

```
$ wget http://download.redis.io/releases/redis-3.2.11.tar.gz
$ tar xzf redis-3.2.11.tar.gz
$ cd redis-3.2.11
$ make
```

If the installation was successful, we can start the Redis server by entering

```
$ src/redis-server
```

from within the `redis-3.2.11` folder. The terminal should display the launch screen and, subsequently, we can shut the server down by pressing `CTRL + c` because we first have to edit its configuration file. We can edit the `redis.conf` file using for example

```
$ nano redis.conf
```

or any other text editor. Line 61 should be commented out so that the server can be accessed from any IP. To restrict access, it is necessary to set a password in line 480. Since Redis quickly processes external requests, strong passwords are needed to impede password guessing. After saving and closing the file, we can start the server using this configuration by entering

```
$ src/redis-server redis.conf
```

when in the installation directory. The **rredis** package can be used to access many of the Redis server functionalities. For example, we can now check whether we can log in to the server. Using the RStudio server, we connect to the Redis server and execute a few commands:

```
> library(rredis)
> redisConnect(password = "myredispw", nodelay = FALSE)
> redisSet("test", runif(10))
> redisGet("test")
```

This should return 10 random numbers.

[†]Connecting to Your Linux Instance from Windows Using PuTTY, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html (accessed on 2018-11-08)
[‡]For more information, see https://redis.io/

2.3 | Step 3: using doRedis with a local worker

To run **doRedis** jobs, we will use the version that is on Github because, based on the author's experience, the `nodelay` argument is needed to avoid errors, which is currently not implemented in the CRAN version. We log in to RStudio, install the **doRedis** package, and run a small job to check whether the **R** session successfully interacts with the Redis server. The function `registerDoRedis()` initiates a work queue named `jobs`, which can later be filled with jobs. The next function `startLocalWorkers()` initiates a number of workers (here, one) that listen on the work queue, the `timeout` and `linger` parameters control for how long the workers listen before they disconnect. After these are initiated, we will start the **foreach** loop to process the tasks. The version 1.2.0 of **doRedis** from https://github.com/bwlewis/doRedis, which was used for this article, is also available as a fork that will not be modified at https://github.com/Thie1e/doRedis.

```
> library(devtools)
> install_github("thie1e/doRedis")
> library("doRedis")
>
> registerDoRedis("jobs", password = "myredispw", nodelay = FALSE)
> startLocalWorkers(n = 1, queue = "jobs",
+                   linger = 24*60*60, timeout = 24*60*60,
+                   nodelay = FALSE, password = "myredispw")
>
> foreach(i = 1:10, .combine = c) %dopar% {
+     control <- rnorm(300000, 50, 1)
+     experimental <- rnorm(300000, 50, 1)
+     t.test(control, experimental, var.equal = TRUE)$p.value
+ }
```

If the **foreach** job returned a vector of p-values and no warnings, **doRedis** and the Redis server successfully calculated the results.

2.4 | Step 4: using doRedis with a remote worker

We then proceed to using **doRedis** with a single remote worker. We start by removing the previously used queue by entering

```
> removeQueue("jobs")
```

in the RStudio server session, which should additionally terminate the local workers. Then, we open a new queue named "jobs2" and start the same `foreach` loop.

```
> removeQueue("jobs")
> registerDoRedis("jobs2", password = "myredispw", nodelay = FALSE)
>
> foreach(i = 1:10, .combine = c) %dopar% {
+     control <- rnorm(300000, 50, 1)
+     experimental <- rnorm(300000, 50, 1)
+     t.test(control, experimental, var.equal = TRUE)$p.value
+ }
```

This time, RStudio should be busy without returning a result because the server is waiting for workers to fetch the job. We can now use a remote worker to connect to this master and fetch and complete the tasks. For this, we will simply start a second instance using our personal AMI.[§] We log in to this new instance, install the necessary packages, and start local

[§]since we will use RStudio Server to connect to both the master and worker simultaneously we recommend using two different browser windows.

workers that connect to the master with:

```
> library(devtools)
> install_github("thie1e/doRedis")
> library("doRedis")
>
> startLocalWorkers(n = 1, queue = "jobs2", host = "[PUBLIC DNS]",
+                   linger = 24*60*60, timeout = 24*60*60,
+                   nodelay = FALSE, password = "myredispw")
```

The value `"[PUBLIC DNS]"` has to be replaced by the actual public DNS address of the master that can be copied and pasted from the AWS management console after selecting the master instance, eg, `ec2-12-34-567-890.eu-central-1.compute.amazonaws.com`. Going back to the master RStudio session where the **doRedis** job was started, we should get the result shortly after. Now, we have, for the first time, initiated a task on a master and connected a worker to it. Note that the master does not actually do any computing work but only distributes the tasks to workers that connect to it and fetch tasks in a specific queue. Since we plan to use a large number of cheaper spot instances as workers, we need to configure workers in such a way that they automatically fetch tasks from one specific master once they are started.

2.5 | Step 5: automatically fetch tasks from a specific Redis-server

We need an **R** script that connects to the **doRedis** job and to start that script automatically as soon as the worker instance is started, if we want to use a spot fleet later on:

```
> myhost <- commandArgs(trailingOnly = TRUE)
> myhost <- as.character(myhost)
>
> if (is.na(myhost)) {
+   stop("Host IP or address is expected as a command line argument")
+ }
>
> library(doRedis)
> startLocalWorkers(n = parallel::detectCores(), queue = "jobs", nodelay = FALSE,
+                   linger = 24*60*60, timeout = 24*60*60,
+                   host = myhost, password = "myredispw")
```

We save this script on the worker via RStudio as `doRedis_startup_myIP.R` in `/home/rstudio/`. To make this script available on future workers, we save the current worker as an AMI, including all of the installed packages and software, by right clicking the instance on the AWS website and clicking “create image” under the “image” category. A few minutes after the request, the new AMI should be available in the “AMI” menu under “Images”.

In order to test whether we can automatically connect using this script, we create a new queue by `registerDoRedis("jobs", password = "myredispw", nodelay = FALSE)` on the host and again start the `foreach` loop. We should not receive a result, yet, because the worker for the queue “jobs” should have shut down. Otherwise, please restart the master using the AWS management console, then restart the Redis server and the `foreach()` loop.

To test the automatic connection, we create a new instance using our personal AMI. In the instance details under “Advanced Details”, we insert the following command as “user data”. This code will be run when the instance is started (not upon stopping and restarting):

```
#!/bin/bash
sudo -H -u rstudio bash -c ′Rscript /home/rstudio/doRedis_startup_myIP.R [PUBLIC DNS]′
```

Here, `[PUBLIC DNS]` has to be replaced by the actual public address of the master, for example, `ec2-12-34-567-890.eu-central-1.compute.amazonaws.com`. Apart from this, we start the instance as before. Eventually, the RStudio session on the master should return the result. If this was successful, we are able to run more complicated tasks on a large number of workers.

2.6 | Step 6: using a spot fleet

As already explained, Amazon offers “spot fleets” that may contain a large number of instances with multiple cores so that it becomes possible to run **R** jobs on practically any number of cores. The fault tolerance of `doRedis` makes it possible to utilize a spot fleet, even if instances are suddenly shut down. Using the same AMI as before, we can now use a large number of workers on extremely lengthy tasks.

First, we have to log in to the master via SSH and start the Redis server. Then, we log in to the master via RStudio and initiate the **doRedis** job and start the loop, which should make the **doRedis** jobs available for the workers. Finally, we start up a spot fleet. We start the job on the master as before:

```
> library("doRedis")
>
> registerDoRedis("jobs", password = "myredispw", nodelay = FALSE)
>
> foreach(i = 1:10, .combine = c) %dopar% {
+     control <- rnorm(300000, 50, 1)
+     experimental <- rnorm(300000, 50, 1)
+     t.test(control, experimental, var.equal = TRUE)$p.value
+ }
```

To start up a spot fleet, click “Spot Requests” under the “Instances” menu in the EC2 management console. In the next window, check “Request and Maintain” to request a spot fleet with a target capacity of the desired number of vCPUs. The AMI has to be our personal AMI with all of the necessary packages to run the job. All instance types that are suitable to run the job can be selected, as AWS will automatically choose the cheapest combination of instances for the spot fleet. As the user data, we have to again enter:

```
#!/bin/bash
sudo -H -u rstudio bash -c ′ Rscript /home/rstudio/doRedis_startup_myIP.R [PUBLIC
DNS]′
```

With the correct host address to make every instance in the spot fleet connect to our Redis server. A few moments after submitting the spot fleet request, we should see the spot fleet's instances in the EC2 management console. Each instance's load can be checked by clicking the instance and selecting the “Monitoring” tab.

Activity on the Redis server can be monitored by opening another SSH session and starting the Redis command line interface by launching

```
$ src/redis-cli.
```

In the CLI enter

```
127.0.0.1:6379 > auth myredispw
```

and then

```
127.0.0.1:6379 > monitor
```

which displays the possibly rapid back and forth of messages between the master and the workers.

2.7 | Step 7: shutting down the instances

Amazon will keep on billing the user for started instances as long as these instances are running, even if these are idle. To shut down the instances, the master instance has to be terminated from the “Instances” menu and the worker instances either have to be terminated from there as well of from the “Spot Requests” menu, if a spot fleet was used. To incur no further costs, instances have to be terminated, not stopped.

3 | A SIMULATION STUDY WITH SEVERAL SCENARIOS

Often, simulation studies are performed to compare different conditions. As a demonstration, take a simple study trying to investigate the effects of heteroscedasticity and sample size imbalance on the classic t-test and the Welch test. Here, a researcher might want to keep the size and variance of one group constant and systematically manipulate the size and variance of the second group. The easiest way to do so is to (1) define a function that takes these values as arguments,

(2) define a list of scenarios, (3) use a loop to go through the different combinations and save the intermediate results to the Dropbox folder, (4) combine the results, and (5) visualize the results. The analysis of the results can be done either using the RStudio server on the master or locally. If using the Rstudio server, by saving to the Dropbox folder, we can ensure that the results cannot be lost by accidentally terminating the master and that they are sent to any other device logged in to the same Dropbox account. The file name is automatically generated based on the scenario's parameters. For the following code, we assume that the **doRedis** queue was already registered to enable parallelization. For preliminary testing and debugging, some repetitions of the job can first be run locally via **doSNOW**.

```
> setwd("~/Dropbox/")
>
> # Part 1: Define function
> ttest_test <- function(samp_exp, var_exp, homvar, repetitions) {
+     foreach(i = 1:repetitions,
+             .errorhandling = "remove",
+             .options.redis = list(chunkSize = 10, progress = TRUE),
+             .combine = rbind) %dopar% {
+                 control <- rnorm(50, 50, 1)
+                 experimental <- rnorm(samp_exp, 50, var_exp)
+                 pval <- t.test(control, experimental,
                    var.equal = homvar)$p.value
+                 result <- cbind(samp_exp, var_exp, homvar, pval)
+             }
+ }
>
> # Part 2: Define scenarios
> repetitions <- 10000
> samplesize <- c(25, 50, 100)
> var_exp <- c(0.2, 1, 5)
> homvar <- c(TRUE, FALSE)
> scenarios <- expand.grid(samplesize = samplesize,
+                          var_exp = var_exp,
+                          homvar = homvar)
>
> # Part 3: Loop over scenarios and repetitions and save results
> for(i in 1:nrow(scenarios)) {
+     res <- ttest_test(scenarios$samplesize[i], scenarios$var_exp[i],
+                 scenarios$homvar[i], repetitions = repetitions)
+     filename <- paste0("pval_t_", scenarios$samplesize[i], "_",
+                 scenarios$var_exp[i], "_", scenarios$homvar[i],
+                 ".RData")
+     save(res, file = filename)
+ }
>
> # Part 4: Combine results
> files <- list.files()
> load(files[1])
> results <- res
> for (i in 2:length(files)) {
+     load(files[i])
+     results <- rbind(results, res)
+ }
>
> # Part 5: Visualize results
```

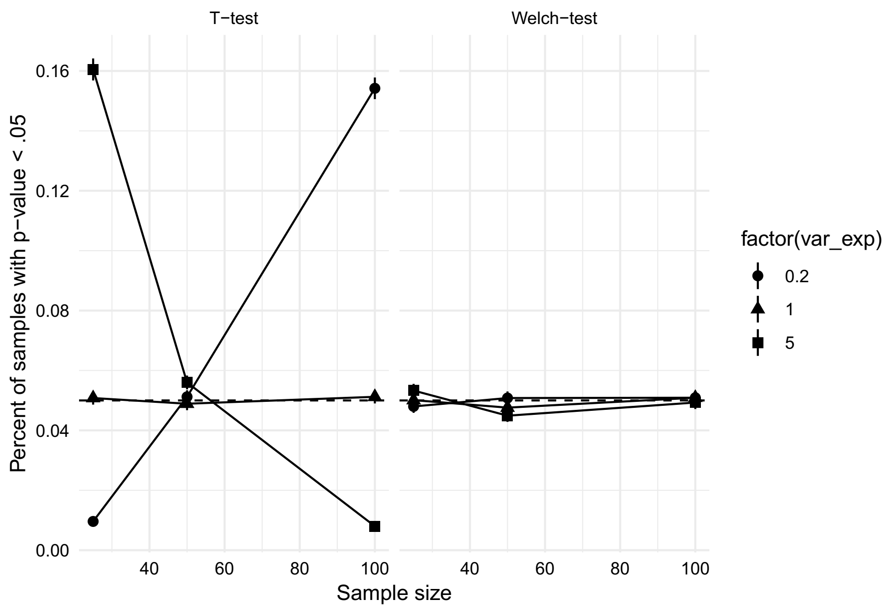

FIGURE 2 Results of the simulation study: (Left) Results for the t-test; (Right) Results for the Welch test

```
> library(ggplot2)
> results <- as.data.frame(results)
> ggplot(results, aes(
+     x = samp_exp,
+     y = as.numeric(pval < 0.05),
+     shape = factor(var_exp))) +
+     stat_summary(geom = "line") +
+     stat_summary() +
+     facet_wrap( ~ homvar) +
+     geom_hline(yintercept = 0.05, linetype=2)+
+     ylab("Percent of samples with p-value < .05") + xlab("Sample size") +
+     theme_minimal()
```

As can be seen in Figure 2 (right), the alpha-error rate for the Welch test hovers around 5% for all scenarios, ie, heteroscedasticity does not affect the alpha-error rate. However, for the t-test (Figure 1 left), heteroscedasticity has a profound effect on the alpha-error rates. Specifically, we see that the alpha-error rates are over 15% if the smaller sample has a larger standard deviation (ie, at a sample size = 25 and SD = 5, or at sample size = 100 and SD = .2). Furthermore, the alpha-error rate is much lower than 5% in scenarios where the smaller sample also has the smaller SD. Heteroscedasticity does not affect the error rates when sample sizes are equal. This has led some authors to argue that the Welch-Test should be used more often.[19,20]

4 | FURTHER CONSIDERATIONS

4.1 | Cost management

Cost management is essential when using larger spot fleets. As long as the requested instances are not terminated, Amazon will continue billing the user. To automatically shut down the spot fleet or single workers, alarms can be set under "Cloud Watch Monitoring" and "Add/Edit Alarms". For example, the instance or fleet can be set to terminate if the maximum CPU utilization does not exceed 50% for at least an hour. Note that the alarm will be immediately triggered if the condition of a certain idle time was met right after starting the instance or fleet. In any case, the workers should be terminated and the spot request deleted after the job is finished to avoid further costs. Tasks that were lost because of instances that were shut down will be resubmitted automatically. The user may have to wait a few minutes for this resubmission to happen, depending on the configuration.

4.2 | Chunking

Another important consideration is the time that is needed to execute a single iteration of the foreach loop. If this time is rather low, the rapid fetching and sending of tasks introduces an overhead that may slow down the process. In that case, `setChunkSize` can be used to increase the number of iterations that a worker will process at once. If the chunk size is too high with respect to the time an iteration takes, this will impair the balancing as some workers may be idle, whereas other workers may still have multiple iterations to finish. One drawback of the default settings of foreach is its behavior to sometimes throw errors only after finishing most of the iterations, which will discard all of the results. As a safer alternative, the user can set the `.errorhandling` argument to `"remove"` as that will remove faulty iterations and, therefore, might return most of the simulation results. Database values in the Redis server are limited to 512 MB. If this limit is exceeded, the data should be broken up into chunks that are processed separately.

4.3 | Reproducibility

An important aspect of simulation studies in the academic context is their reproducibility, ie, the ability to get the exact same results when the simulation study is repeated.[3] For reproducibility, the random number generation can be initiated via the familiar `set.seed` when using **doRedis**. This assigns every foreach loop iteration a L'Ecuyer stream to achieve reproducible parallel random number generation. User-defined random seed generation functions can be used as well but the standard method should usually suffice.

5 | DISCUSSION

The aim of this paper was to give a hands-on introduction to cloud-based parallelization techniques in R to enable more efficient simulation studies in medicine.[3] Specifically, we provided a step-by-step tutorial to setting up and using a fleet of Amazon's spot instances. By way of an example simulation study into the effects of heteroscedasticity on the equal-variance t-test and Welch-test, we were able to show how typical simulation studies are performed in **R**.

We are sure that alternative methods will develop in the future that may speed up or end the need to set up AWS clusters manually. For example, the **aws.ec2** package aims to allow for initiating, starting, and stopping AWS instances. The **doAzureParallel** package[21] is another example for a project that tries to leverage the parallel computations for R but relies on Microsoft Azure instead of AWS for cloud computing capacities. The configuration is simpler and requires fewer steps and less testing than the procedure described in this article. It allows for starting cloud instances from a local **R** session, automatically scales back the number of nodes if there is no load, and, due to its fault tolerance, enables the use of low priority VMs that are offered at a discount, similar to spot instances. The package is in an early stage and not yet on CRAN (April 2018) but seems to be a promising addition to cloud computing facilities in **R**. As a result, there is currently no easy-to-use alternative to set up cheap, reproducible, and large-scale simulation studies in R. Furthermore, we believe that this tutorial provides a good conceptual introduction to the issues relevant to running parallel simulation studies in R, even though some of the steps may become obsolete in the future.

An important question in setting up these cloud-based studies is whether or not it is worth the effort. Realistically, it may take a day to successfully set up and test the Redis server, an AMI with all of the necessary packages, and a suitable simulation script. It may be more time-effective to simply rent the largest AWS-instance and use the much more convenient local parallelization techniques. However, whenever it is essential to cut overall costs or reduce the time by increasing the number of CPUs that work in parallel, the procedure described here may give an edge over local and more simple cloud-based methods.[22]

ACKNOWLEDGEMENT

The study was supported by the German Federal Ministry for Education and Research (BMBF #01EK1501).

ORCID

G. Hirschfeld https://orcid.org/0000-0003-2143-4564
C. Thiele https://orcid.org/0000-0002-1156-5117

REFERENCES

1. Morgan BJT. *Elements of Simulation*. Boca Raton, FL: CRC Press; 1984.
2. Ripley BD. *Stochastic Simulation*. New York, NY: John Wiley & Sons; 2009.
3. Burton A, Altman DG, Royston P, Holder RL. The design of simulation studies in medical statistics. *Statist Med*. 2006;25(24):4279-4292.
4. Maxwell SE, Cole DA. Tips for writing (and reading) methodological articles. *Psychol Bull*. 1995;118(2):193-198.
5. Hodgson T, Burke M. On simulation and the teaching of statistics. *Teach Stat*. 2000;22(3):91-96.
6. R Core Team. R: A language and environment for statistical computing. R Foundation for Statistical Computing. 2017.
7. Hallgren KA. Conducting simulation studies in the R programming environment. *Tutor Quant Methods Psychol*. 2013;9(2):43-60.
8. Foster I. *Designing and Building Parallel Programs*. Boston, MA: Addison Wesley Publishing Company Reading; 1995.
9. Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U. *State-of-the-Art in Parallel Computing with R*. Technical Report. Munich, Germany: Ludwig Maximilian University of Munich; 2009.
10. Delgado MS, Parmeter CF. Embarrassingly easy embarrassingly parallel processing in R. *J Appl Econ*. 2013;28(7):1224-1230.
11. Eugster MJA, Knaus J, Porzelius C, Schmidberger M, Vicedo E. Hands-on tutorial for parallel computing with R. *Comput Stat*. 2011;26(2):219-239.
12. Student. Probable error of a mean. *Biometrica*. 1908;6(1):1-25.
13. Welch BL. The significance of the difference between two means when the population variances are unequal. *Biometrica*. 1938;29:350-362.
14. Analytics R, Weston S. doSNOW: Foreach parallel adaptor for the snow package. R Package Version 1. 2014.
15. Analytics R, Weston S. Foreach: provides foreach looping construct for R. R Package Version 1.4.3. 2015.
16. Rabl T, Gómez-Villamor S, Sadoghi M, Muntés-Mulero V, Jacobsen HA, Mankovskii S. Solving big data challenges for enterprise application performance management. *Proc VLDB Endow*. 2012;5(12):1724-1735.
17. Lewis B. doRedis: Foreach parallel adapter for the rredis package. R Package Version 1.2.0. 2015.
18. Ben-Yehuda OA, Ben-Yehuda M, Schuster A, Tsafrir D. Deconstructing Amazon ec2 spot instance pricing. *ACM Trans Econ Comput*. 2013;1(3): Article 16.
19. Ruxton GD. The unequal variance *t*-test is an underused alternative to Student's *t*-test and the Mann–Whitney *U* test. *Behav Ecol*. 2006;17(4):688-690.
20. Delacre M, Lakens D, Leys C. Why psychologists should by default use Welch's *t*-test instead of student's *t*-test. *Int Rev Soc Psychol*. 2017;30(1):92-101.
21. Hoang B. doazureparallel. https://github.com/Azure/doAzureParallel. 2018.
22. Knaus J, Hieke S, Binder H, Schwarzer G. Costs of cloud computing for a biometry department. *Methods Inf Med*. 2013;52(01):72-79.

How to cite this article: Hirschfeld G, Thiele C. Cloud-based simulation studies in **R** - A tutorial on using doRedis with Amazon spot fleets. *Statistics in Medicine*. 2019;38:3947–3959. https://doi.org/10.1002/sim.8188