Parallel computing with R [1]

Matthieu Bruneaux

October 7, 2014

[1] Git commit hash: 38bf894b77fab21e40560028d82b1adc91c7e044

Contents

Chapter 1

Parallel computing with R

1.1 When can parallel computing be used?

Parallel computing enables to do calculations simultaneously on several cores. If a desktop computer has a multi-core processor, it usually uses only one core at a time to run an R session. However, for certain analyses, it is possible to take advantage of the presence of several cores and to run independent parts of the analysis on different cores, simultaneously (figure 1.1).

Not all analyses can be performed using parallel computing. Some analyses require sequential steps, and each step depends on the result of the previous one. In this case, it is not possible to distribute the calculation load on independent processes. However, many analyses with heavy calculation requirements consist of a few analysis steps which are repeated a large number of time, independently, on different inputs. Here are some examples:

- G_{st} values are available along the genome for your favourite species. You want to perform a kernel smoothing and some permutation testing to detect regions with significantly high Gst on each chromosome. The analyses for each chromosome are independent and can be performed on separate cores.

- You want to test the robustness of your results by bootstrapping. You have to rerun your analysis a large number of time on resampled datasets produced by shuffling the original dataset. Each analysis on a resampled dataset is independent from the others, and the bootstrap simulations can be distributed among several cores.

After the parallel step, another step is often necessary to gather the results produced independently (e.g. concatenate the results for each chromosome into a single table or pool the results of all the bootstrap simulations to calculate p-values). However, the most time-consuming step is usually the one which can be distributed on multiple cores.

One quick-and-dirty way to use several cores with R is to manually start as many R sessions as available cores. However, when scaling an analysis to 16, 32 or more cores on a computer grid, manually starting and managing R scripts is not a good option. A much more simple, robust and elegant solution is to use one of the R packages for parallel computing.

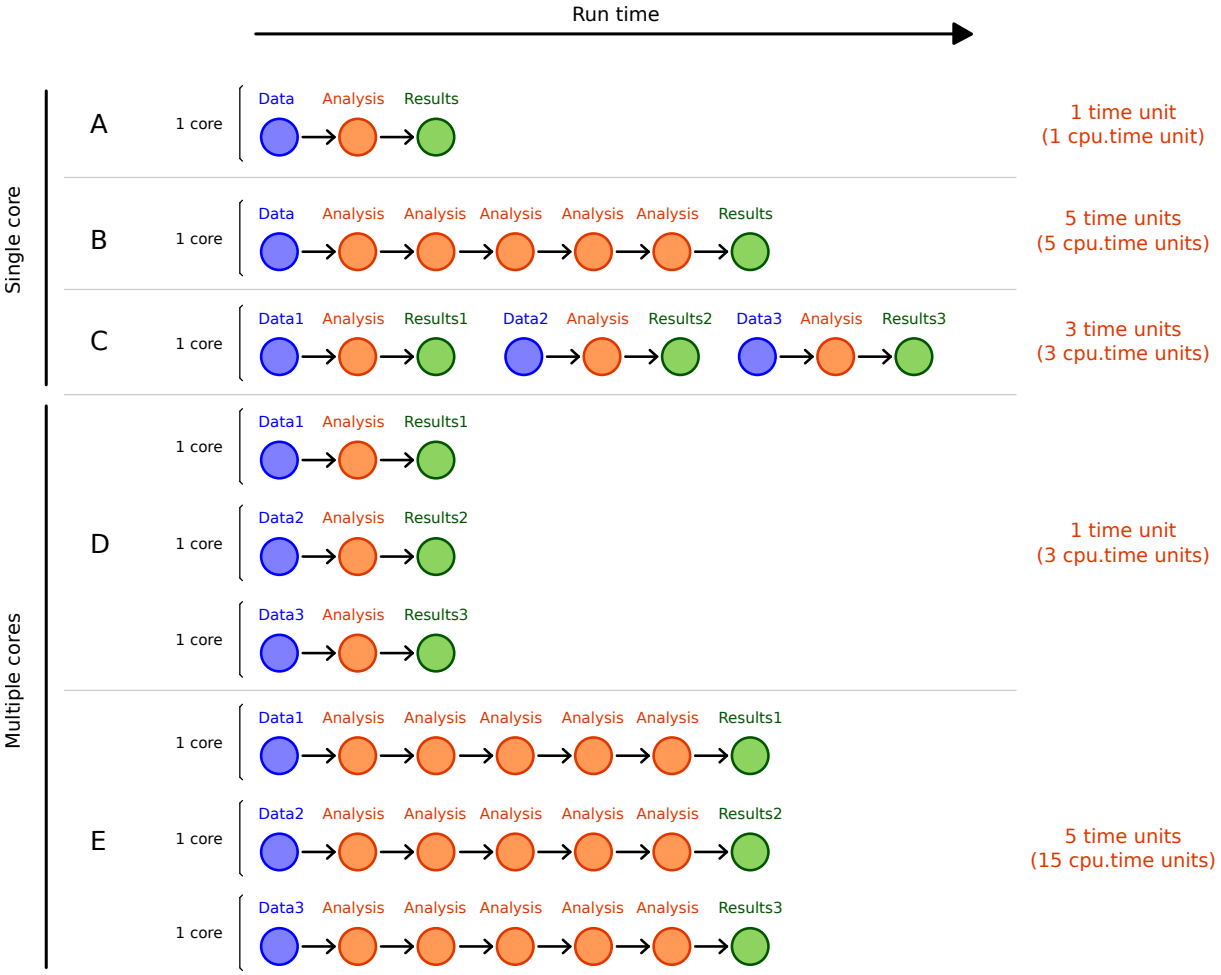

Figure 1.1: Comparison of performances between single and multiple core runs. Each orange circle represents one cpu.time unit (e.g. the amount of calculations performed by one core during one hour). In A, a simple analysis needs one cpu.time unit to be run (e.g. a one hour run with one core). In B, a longer analysis is performed, which needs 5 cpu.time units (e.g. a five hour run with one core). In this case, each step of the analysis depends on the previous one, and the run has to be sequential. In C, however, the analysis is the same as in A but is performed on a larger amount of input data. The analysis requires three cpu.time units (the three orange circles, e.g. a three hour run with one core). Since the analysis performed in C is actually independent for each data packet, it can be parallelized as shown in D and run in only one time unit if three cores are used (e.g. a one hour parallel run with three cores). The amount of calculations performed is the same as in C (three cpu.time units), but the elapsed time is shorter since three cores were working at the same time. The sequential analysis in B cannot be made faster by parallelization since each analysis step depends on the previous step (true sequential analysis), but if the analysis pipeline has to be run on several data packets independently then it can be parallelized to make the analysis run faster when there is more data as shown in E (e.g a five hour run with three cores instead of a 15 hour run with one core).

1.2 Approach to parallel computing with R used in this document

The aim of this document is to provide a step-by-step guide of how to use parallel computing with R. At the end of this tutorial, you should be able to take advantage of the multiple cores of your laptop or of your desktop computer, or to use 16, 32 or more cores on the CSC server Taito (http://research.csc.fi/taito-user-guide) for larger parallel runs.

A lot of detailed resources concerning parallel computing with Rare already available and easily found on the Internet. One can look through the documentation of the corresponding packages or have a look on GitHub and search for 'parallel R'.

1.2.1 Packages and resources for parallel computing with R

A list of useful packages can be found on the CRAN webpage concerning R and high performance computing. There are many different options; I will mainly focus on the snow package since I managed to use this one both on a CSC server (Taito) and on my desktop computer and its setup is quite simple. The approach described here with snow works well with a personal computer or when using a single node on Taito (each Taito node contains 16 cores), but for analyses using multiple nodes (e.g. 32 or 64 cores) we will use the Rmpi package.

1.2.2 Using *apply* functions in R code

The approach explained in this document relies on the use of the *apply* family of functions in R. Those functions are apply, sapply and lapply and are part of the base package. They are seldom used by R users when they just start to learn R, but they are of considerable interest when one becomes more familiar with the language. The documentation can be accessed by typing ?apply or ?lapply in R.

Each function of this family takes as an input an object made of several elements (e.g. a vector or a list) and applies a function on each element of this object. Here is a very simple example using a vector of numbers, and returning the square of each number:

```
 # vector object with four elements
 v = c(1, 2, 3, 4)
 # show v
 v
[1] 1 2 3 4
 # function returning the square of its input
 f = function(x) { x^2 }
 # show f
 f
function(x) { x^2 }
 # example of f usage
 f(5)
[1] 25
 f(6)
[1] 36
 # sapply can apply the f function to each element of v
 sapply(v, f)
[1]  1  4  9 16
```

```
# now we want to apply f to the integers from 5 to 15
# a range of integers can be specified using the ":" notation
5:15
[1]  5  6  7  8  9 10 11 12 13 14 15
sapply(5:15, f)
[1]  25  36  49  64  81 100 121 144 169 196 225
```

`sapply`, `lapply` and `apply` differ in the type of objects on which they work:

- `sapply` will apply a function on each element of a *vector*
- `lapply` on each element of a *list*
- `apply` on each row or each column of a *matrix*

Since those functions work on each element of an object independently from the other elements, they are easily amenable to parallelization. This is done in a very simple manner with the `snow` package: the function names just have to be replaced with the corresponding `parSapply`, `parLapply` or `parApply` parallelized function names.

An efficient use of `sapply`, `lapply` and `apply` requires a good understanding of the `vector`, `list` and `matrix` or `data.frame` objects. In particular, we will be using the `list` object which is very versatile but might seem a bit daunting to users without any prior experience with it.

Using the *apply* family of functions tends to produce cleaner and better code - cleaner since the user has to structure the data efficiently into e.g. lists and to divide its code into separate functions, and better because the code is easier to read and to debug - and is of interest even for code which is not intended to be used on multiple cores.

1.2.3 Refactoring code for parallel runs

Running code which already uses any *apply* family function on multiple cores requires only a few modifications, such as setting up the core cluster at the beginning of the script and converting the *apply* calls to the corresponding *parApply* calls.

Running code which is already working fine but without using any *apply* function yet requires a bit more work to refactor it before being able to use it on multiple cores, but even then the amount of work is relatively minor and involves mainly a wrapping of the analysis into a main function and a storage of the data into a suitable list structure.

1.3 What will be done in this tutorial

This tutorial will focus on using the `list` object to store data and the `lapply` and `parLapply` functions to apply a function on each element of a list.

The tutorial is organized as follows:

- The first part is about the `list` object structure and how to use it.
- The second part will describe how to run a script using the `snow` package on several cores on a local computer.
- The third part will show how to run the same parallel code on CSC servers, using up to 16 cores on Taito.

- The fourth part will explain how to use more than 16 cores on CSC servers using the `Rmpi` package.

Chapter 2

R basics for parallel code

This section describes the basics needed prior to running code on multiple cores using data stored as a list and the `lapply` approach. It covers list creation and manipulation and `lapply` call.

If you are already familiar with lists and `lapply`, you can skip it and go directly to section 3.

2.1 Basic list manipulation

Lists can be seen as sophisticated R vectors. A vector in R can store a sequence of objects of the same type (e.g. all numbers or all strings), but cannot store objects of different types. A list can store objects of different types. Below is a short introduction about vectors and lists in R.

A vector can be created using the `c()` function to combine values.

```
 x = c(1, 4, 2, 5)
 x
[1] 1 4 2 5
 class(x)
[1] "numeric"
```

A vector can store numbers or strings.

```
 x = c(1, 4, 2, 5)
 x
[1] 1 4 2 5
 class(x)
[1] "numeric"
 y = c("sweave", "R", "latex", "knitr")
 y
[1] "sweave" "R"      "latex"  "knitr"
 class(y)
[1] "character"
```

An element from a vector can be accessed with the `[]` notation.

```
 y
[1] "sweave" "R"      "latex"  "knitr"
 y[1]
[1] "sweave"
 y[2:4]
```

```
[1] "R"      "latex" "knitr"
 y[c(2, 4)]
[1] "R"      "knitr"
```

However, a vector cannot mix different types of objects:

```
 z = c(1, 4, "sweave", "R")
 z
[1] "1"      "4"      "sweave" "R"
 class(z)
[1] "character"
```

If we try to mix numbers and strings in a single vector, R automatically convert the numbers to strings.

A list can store a sequence of objects as vectors do, but those objects can be of different types. A list can be created using the `list()` function.

```
 lst = list(1, 4, "sweave", "R")
 class(lst)
[1] "list"
```

An element from a list can be accessed using the `[[]]` notation.

```
 length(lst)
[1] 4
 lst[[1]]
[1] 1
 lst[[3]]
[1] "sweave"
```

A list does not modify the type of its elements.

```
 class(lst[[1]])
[1] "numeric"
 class(lst[[3]])
[1] "character"
```

An interesting feature of lists is that each element can be named and retrieved by its name. To retrieve an element by its name, we can use the `[[]]` or `$` notation.

```
 lst = list("player" = c("Socrates", "Karl Marx"),
            "goals" = c(2, 3))
 str(lst) # str() displays the structure of an object
List of 2
 $ player: chr [1:2] "Socrates" "Karl Marx"
 $ goals : num [1:2] 2 3
 lst[["player"]]
[1] "Socrates"  "Karl Marx"
 lst$goals
[1] 2 3
```

Elements from a named list can still be retrieved by their index.

```
 names(lst)
[1] "player" "goals"
 lst[[1]]
[1] "Socrates"  "Karl Marx"
 lst[[2]]
[1] 2 3
```

Elements can be added after the creation of the list.

```
 lst[["team"]] = c("Greece", "Germany")
 names(lst)
[1] "player" "goals"  "team"
 lst$team
[1] "Greece"  "Germany"
 lst[[3]]
[1] "Greece"  "Germany"
```

A new named element is automatically appended after the last element. An element can also be added using an index number.

```
 lst[[4]] = c("wine", "beer")
 lst[[7]] = c("Athens", "Berlin")
 str(lst)
List of 7
 $ player: chr [1:2] "Socrates" "Karl Marx"
 $ goals : num [1:2] 2 3
 $ team  : chr [1:2] "Greece" "Germany"
 $       : chr [1:2] "wine" "beer"
 $       : NULL
 $       : NULL
 $       : chr [1:2] "Athens" "Berlin"
```

If an element is added with an index number greater than the last element index plus one, `NULL` objects are automatically inserted in between.

The names of the elements of a list and the elements themselves can also be modified afterwards.

```
 names(lst)
[1] "player" "goals"  "team"   ""       ""       ""       ""
 names(lst)[4] = "beverage"
 names(lst)[7] = "capital.city"
 str(lst)
List of 7
 $ player      : chr [1:2] "Socrates" "Karl Marx"
 $ goals       : num [1:2] 2 3
 $ team        : chr [1:2] "Greece" "Germany"
 $ beverage    : chr [1:2] "wine" "beer"
 $             : NULL
 $             : NULL
 $ capital.city: chr [1:2] "Athens" "Berlin"
 lst[[7]][2] = "Trier"
 names(lst)[c(4, 7)] = c("preferred.beverage", "birth.place")
 str(lst)
List of 7
 $ player            : chr [1:2] "Socrates" "Karl Marx"
 $ goals             : num [1:2] 2 3
 $ team              : chr [1:2] "Greece" "Germany"
 $ preferred.beverage: chr [1:2] "wine" "beer"
 $                   : NULL
 $                   : NULL
 $ birth.place       : chr [1:2] "Athens" "Trier"
```

Lists are sequences of objects, and as such it is possible to iterate over each object contained in a list.

```
 for (object in lst) {
   print(object[1]) # print the first element of each object
 }
[1] "Socrates"
[1] 2
[1] "Greece"
[1] "wine"
NULL
NULL
[1] "Athens"
```

2.2 Using `sapply` and `lapply` on a vector and on a list

In the introduction, we used `sapply` to calculate the square for numbers contained in a vector. The return value is also a vector:

```
 # define the vector
 v = 1:10
 v
 [1]  1  2  3  4  5  6  7  8  9 10
 # function to return the square
 f = function(x) { x^2 }
 # apply f to each element of v with sapply
 r = sapply(v, f)
 r
 [1]   1   4   9  16  25  36  49  64  81 100
 # each element of r corresponds to an element of v
 # for example
 v[2]
[1] 2
 r[2] # r[2] is f(v[2])
[1] 4
 v[5]
[1] 5
 r[5] # r[5] is f(v[5])
[1] 25
```

To apply a function to each element of a list, we use `lapply`. The return value is a list:

```
 # define a list
 lst = list()
 lst[["length_male_dragon"]] = c(7, 8, 6, 8, 7, 6)
 lst[["length_female_dragon"]] = c(5, 6, 7, 5, 6, 8)
 str(lst)
List of 2
 $ length_male_dragon  : num [1:6] 7 8 6 8 7 6
 $ length_female_dragon: num [1:6] 5 6 7 5 6 8
 # define a function to calculate the mean length
 f = function(x) { sum(x) / length(x) }
 # apply f to each element of lst
 r = lapply(lst, f)
 class(r) # r is a list also
[1] "list"
 str(r) # each element of r matches an element of lst
List of 2
 $ length_male_dragon  : num 7
 $ length_female_dragon: num 6.17
```

```
 lst[[1]]
[1] 7 8 6 8 7 6
 r[[1]] # r[[1]] is f(lst[[1]])
[1] 7
 lst[[2]]
[1] 5 6 7 5 6 8
 r[[2]] # r[[2]] is f(lst[[2]])
[1] 6.166667
```

In the next section (3), we will see more meaningful examples of how to store data into a list from an existing dataset and of using `lapply`.

2.3 Basic data frame manipulation

Running parallel code often requires to store an existing dataset (saved in a text file from Excel and loaded into `R` using `read.table` for example, or stored into separate files on the disk) into a list, dividing the original dataset into separated list elements.

When a data set is loaded from a text file, it is usually stored as a *data frame*. `iris` is a dataset shipped with R that we will use in this short example. It is a data frame, which is equivalent to a table containing data arranged in columns. Each row represents a record and each column represents a variable.

```
 class(iris) # data frame
[1] "data.frame"
 dim(iris) # number of rows (records) and of columns (variables)
[1] 150   5
```

The names of the columns can be retrieved easily:

```
 names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "
   Species"
```

We can have a look at the data structure:

```
 head(iris) # show the first rows to have an idea of the table content
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
 str(iris) # show the type of each column
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1
    1 1 1 1 1 1 ...
```

The data frame `iris` contains 150 records for which 5 variables are available. The variables are `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width` and `Species`.

Data frames and lists are actually very similar in the way the data can be accessed. Data frame columns are equivalent to list elements, and can be accessed using the same notations: `[[]]` and `$`.

```
 # we use head to show only the beginning of each object
 head(iris[[1]]) # Sepal.Length column
[1] 5.1 4.9 4.7 4.6 5.0 5.4
 head(iris[["Sepal.Width"]])
[1] 3.5 3.0 3.2 3.1 3.6 3.9
 head(iris$Petal.Length)
[1] 1.4 1.4 1.3 1.5 1.4 1.7
```

Another very convenient way to access the data in a dta frame is to use the `[row, column]` notation:

```
 # element from the first row, first column
 iris[1, 1]
[1] 5.1
 # element from the first row, 4th column
 iris[1, 4]
[1] 0.2
 # element from the second row, third column
 iris[2, 3]
[1] 1.4
```

Integer ranges are allowed in the `[,]` notation:

```
 # elements from rows 1 to 3, first column
 iris[1:3, 1]
[1] 5.1 4.9 4.7
 # elements from rows 2 to 4, columns 2 and 3
 iris[2:4, 2:3]
  Sepal.Width Petal.Length
2         3.0          1.4
3         3.2          1.3
4         3.1          1.5
```

Any vector of integers is actually allowed:

```
 # elements from rows 1, 4, 5, and third column
 iris[c(1, 4, 5), 3]
[1] 1.4 1.5 1.4
 # elements from rows 4, 1 and columns 1 and 2
 iris[c(4, 1), c(1, 2)]
  Sepal.Length Sepal.Width
4          4.6         3.1
1          5.1         3.5
```

Whole rows or whole columns can be selected if one side of the comma is left blank:

```
 # select ALL the rows in the 4th column
 a = iris[, 4]
 length(a)
[1] 150
 head(a)
[1] 0.2 0.2 0.2 0.2 0.2 0.4
 # select ALL the rows in the 3th and 5th columns
 b = iris[, c(3,5)]
 dim(b)
[1] 150   2
 head(b)
```

```
  Petal.Length Species
1          1.4  setosa
2          1.4  setosa
3          1.3  setosa
4          1.5  setosa
5          1.4  setosa
6          1.7  setosa
 # select ALL the columns for records 1 to 5
 c = iris[1:5, ]
 c
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
```

Finally, one of the most useful notation is to use conditions to select a set of rows or columns. First we can store in a vector (`short.sepal`) whether or not the sepal length of each record is less than 5:

```
 short.sepal = (iris$Sepal.Length < 5)
 length(short.sepal)
[1] 150
 head(iris$Sepal.Length)
[1] 5.1 4.9 4.7 4.6 5.0 5.4
 head(short.sepal)
[1] FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

The resulting vector contains `TRUE` and `FALSE` values indicating whether or not the condition `(iris$Sepal.Length < 5)` was true. This vector can be used to specify which rows of the `iris` data frame we want to select. We do this by using the TRUE/FALSE vector to specify the rows in the `[,]` notation, and by leaving the column indication blank to retrieve all the columns for those rows:

```
 short.sepal.iris = iris[short.sepal, ]
 dim(short.sepal.iris)
[1] 22  5
 head(short.sepal.iris)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
7           4.6         3.4          1.4         0.3  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
```

22 records with short sepals were retrieved.

2.4 Using `apply` on a data frame

`apply` is the *apply* family function that can be used to apply one function on every row or every column of a data frame. A call to `apply` must provide the input data frame, specify if it has to work on rows or columns, and provide the function to apply to each row or column.

For example, we can calculate the mean and standard deviation for each column of the `iris` dataset:

```
 # we use only the first four columns
 head(iris[, 1:4])
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
 # in this apply call, 2 specifies to work on columns, and mean
 # is the function we apply to each column
 apply(iris[, 1:4], 2, mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
 # sd calculates standard deviation
 apply(iris[, 1:4], 2, sd)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

Let's build a data frame for which we will calculate the row means:

```
 # prepare an empty matrix
 a = matrix(nrow = 3, ncol = 5)
 # fill the rows with random numbers
 a[1, ] = rnorm(5, 7) # 5 samples from a normal distribution with mean 7
 a[2, ] = rnorm(5, 5) # 5 samples from a normal distribution with mean 5
 a[3, ] = rnorm(5, 2) # 5 samples from a normal distribution with mean 2
 a
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 8.262954 6.673767 8.329799 8.272429 7.414641
[2,] 3.460050 4.071433 4.705280 4.994233 7.404653
[3,] 2.763593 1.200991 0.852343 1.710538 1.700785
 # convert the matrix to a data frame
 a = as.data.frame(a)
 a
        V1       V2       V3       V4       V5
1 8.262954 6.673767 8.329799 8.272429 7.414641
2 3.460050 4.071433 4.705280 4.994233 7.404653
3 2.763593 1.200991 0.852343 1.710538 1.700785
 # apply the mean function to each row
 apply(a, 1, mean) # 1 specifies to work on rows
[1] 7.790718 4.927130 1.645650
```

2.5 Example: arranging data into a list and using `lapply`

In this example, we assume that our data is stored in a unique data frame, `iris`. This data frame contains data for several different species of iris, and we would like to perform an analysis for each species. To do so, we will split the data from the data frame into species specific data blocks, stored in a list, and then apply an analysis function to each element of the list with `lapply`.

2.5.1 Splitting the data frame into a list

There are three different species in the `iris` dataset:

```
 unique(iris$Species)
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

We will separate the data for each species into three elements stored in one list. First, let's create an empty list:

```
 sp.data = list()
```

For each species, we now store the corresponding subset from the `iris` dataset into the list with a `for` loop:

```
 species = unique(iris$Species) # we retrieve the  species names
 species
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
 # now we go through a for loop
 for (sp in species) {
   # sp takes the values in species successively
   # during the first iteration, sp is setosa
   # during the second iteration, sp is versicolor
   # during the third iteration, sp is virginica
   #
   species.rows = (iris$Species == sp) # condition on Species
   # species.rows now contains TRUE/FALSE indicating the match
   # between the value of sp and the iris$Species column
   #
   sp.data[[sp]] = iris[species.rows, ]
   # now we store the corresponding rows from the iris data frame,
   # taking all the columns, into sp.data, in an element with the
   # species name contained in sp
 }
```

The list should now have three elements, each element being a data frame corresponding to a given species.

```
 names(sp.data)
[1] "setosa"     "versicolor" "virginica"
```

Let's check *setosa*:

```
 head(sp.data[["setosa"]])
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

And it is the same for *versicolor* and *virginica*:

```
 head(sp.data[["versicolor"]])
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
51          7.0         3.2          4.7         1.4 versicolor
52          6.4         3.2          4.5         1.5 versicolor
53          6.9         3.1          4.9         1.5 versicolor
54          5.5         2.3          4.0         1.3 versicolor
55          6.5         2.8          4.6         1.5 versicolor
56          5.7         2.8          4.5         1.3 versicolor
 head(sp.data[["virginica"]])
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
101          6.3         3.3          6.0         2.5 virginica
102          5.8         2.7          5.1         1.9 virginica
103          7.1         3.0          5.9         2.1 virginica
104          6.3         2.9          5.6         1.8 virginica
105          6.5         3.0          5.8         2.2 virginica
106          7.6         3.0          6.6         2.1 virginica
```

2.5.2 Applying a function to each element of a list with `lapply`

We want to calculate the average petal length for each iris species. For this, we first define a function that will calculate the average value for the column `Petal.Length` in any data frame. This time we use the built-in `mean` function instead of `sum` and `length`.

```
 calculate.mean.petal.length = function(x) {
   mean(x$Petal.Length)
 }
```

We can use `lapply` to apply the function on each element of the list that we built previously.

```
 # apply the function to each element of the list
 mean.lengths = lapply(sp.data, calculate.mean.petal.length)
 # display the results
 mean.lengths
$setosa
[1] 1.462

$versicolor
[1] 4.26

$virginica
[1] 5.552
```

Chapter 3

Parallel computing on a local computer (using the `snow` library) [1]

Parallel computing produces results in a shorter time not by calculating faster, but by running several independent calculations at the same time. If we make an analogy between a car factory and an R script, parallelization speeds up the car production not by making the assembly line work faster, but by providing several assembly lines working at the same speed in parallel.

In this section we provide an example of parallel computing for bootstrap analysis, using the `snow` library.

3.1 Reminder: steps to write parallel code

For an R script to be run in parallel, several steps are needed:

- The data must be stored as elements of a list.
- The analysis must be contained into a function that can be applied (independently) on each element of this list.

We have seen a simple example covering those two points in the previous section. The next steps are:

1. The R script must be able to control several cores.
2. Each core must be given the information needed (i.e. variables and functions).
3. A parallel version of `lapply` must distribute the calls of the analysis function to each element of the data list on the different cores.
4. Depending on the output of the analysis function, a last step might involve collecting the results from parallel processes into a single object.

For a bootstrap analysis to be run in parallel, two different approaches can be used:

[1] The examples given in this section assume a good knowledge of the R basics, including loops, linear models and plotting. However, even if you only know the basics which were introduced in the previous section, you should be able to grasp the general ideas presented here.

- Generate bootstrap resamples, store them in a list and apply the analysis function on each element of the list to obtain a parameter distribution for the bootstrap resamples (this is the approach we will use here).
- Store indices in a list (e.g. from 1 to 10000), and apply the analysis function to them. In this case the bootstrap resample generation is done within the analysis function, and the memory needs are reduced.

3.2 Basic cluster setup

3.2.1 Example analysis

We will illustrate the basic setup for parallel computing on a local computer by bootstrapping regression models using the `cars` dataset.

The `cars` dataset contains the speed and stopping distance for several cars and was recorded in the 1920s (see `?cars`). The kinetic energy of a car is $\frac{1}{2}mv^2$. We can expect the stopping distance to be proportional to the amount of kinetic energy that has to be dissipated by the brakes, which means that there should be a linear relationship between the square of the speed v^2 and the stopping distance d (if we assumes the cars weighed the same).

```
 head(cars)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
 # calculate v^2
 cars$speed.squared = cars$speed ^ 2
 # fit a linear regression on the observed data
 model.observed = lm(cars$dist ~ cars$speed.squared)
```

We want to estimate the 95% confidence interval for the slope of the linear regression model. To do this, we perform a bootstrap:

1. We generate n bootstrap resamples by sampling with replacement from the original dataset.
2. For each bootstrap resample, we calculate the slope of the linear regression model.
3. Based on the distribution of those n slopes, we determine the 95% confidence interval of the real slope.

3.2.2 Strategy

We need to apply the regression model on many bootstrap resamples. In this case, each linear regression on a bootstrap resample is independent from the other linear regressions. This independence means that the analyses can be run in parallel. The steps we are going to follow are:

1. Store the input data (the bootstrap resamples) into a list.

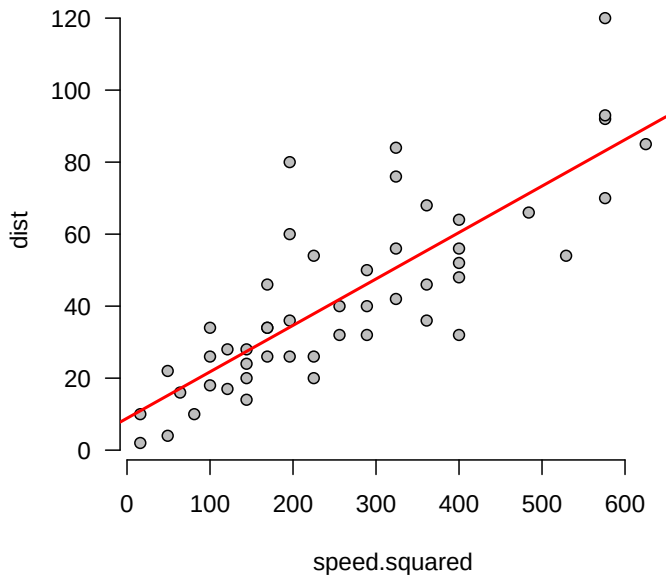

Figure 3.1: Relation between car speed (speed.squared) and stopping distance (dist), taken from the `cars` dataset. The red line is the linear model fit.

2. Create a function which will perform the analysis (linear regression) on each input dataset.

3. Apply the analysis function to each input dataset in parallel using `parLappply`.

4. Process the results.

3.2.3 Preparing bootstrap resamples

500 bootstrap resamples are generated and stored into a list.

```
 n.bootstrap = 500
 # create an empty list
 resamples = list()
 # generate the resamples
 for (i in 1:n.bootstrap) {
   # first draw randomly the row indices for the resample
   resampled.rows = sample(1:nrow(cars), replace = T)
   # then store the resampled data into the list
   resamples[[i]] = cars[resampled.rows, ]
 }
 # we now have our bootstrap resamples ready
 length(resamples)
[1] 500
 head(resamples[[1]])
    speed dist speed.squared
18    13   34           169
25    15   26           225
30    17   40           289
```

```
25.1    15   26          225
10      11   17          121
42      20   56          400
 # the first column is the original row number in the cars
 # data frame. A ".1" is appended when the row is sampled
 # multiple times in a resampled set.
```

3.2.4 Defining the function for analysis

The function used for the analysis of each resample is very simple: it fits a linear model to the resample and returns the linear model.

```
linear.fit = function(resample.cars) {
  # calculate speed.squared
  resample.cars$speed.squared = resample.cars$speed ^ 2
  # linear model
  model.resample=lm(resample.cars$dist ~ resample.cars$speed.squared)
  # return the model
  model.resample
}
```

3.2.5 Setting up the cluster and running the bootstrap

To use multiple cores, we set up a cluster using the `snow` package.

```
library(snow)
# define the number of cores
n.cores = 3
# create the cluster with as many cores as needed
cluster = makeSOCKcluster(rep("localhost", n.cores))
```

The cluster is now ready. We can run the analysis in parallel using the `parLapply` function. The usage is the same as for `lapply`, except that we have to specify the cluster as the first argument. The output is also a list.

```
# run the bootstrap on the cluster
bootstrap.models = parLapply(cluster, resamples, linear.fit)
```

When the run is done, we have to stop the cluster properly:

```
stopCluster(cluster)
```

3.2.6 Examining the results

Let's extract the slopes from the models:

```
# extract the slopes from the bootstrapped models with lapply
# (we define a function on the fly to extract the slopes, which
#  are stored in model$coefficients[2] for each model)
bootstrap.slopes = lapply(bootstrap.models,
                          function(x) x$coefficients[2])
```

The output of `lapply` is a list. We can convert it to a vector using `unlist` and dropping the names of the elements.

```
 class(bootstrap.slopes)
[1] "list"
```

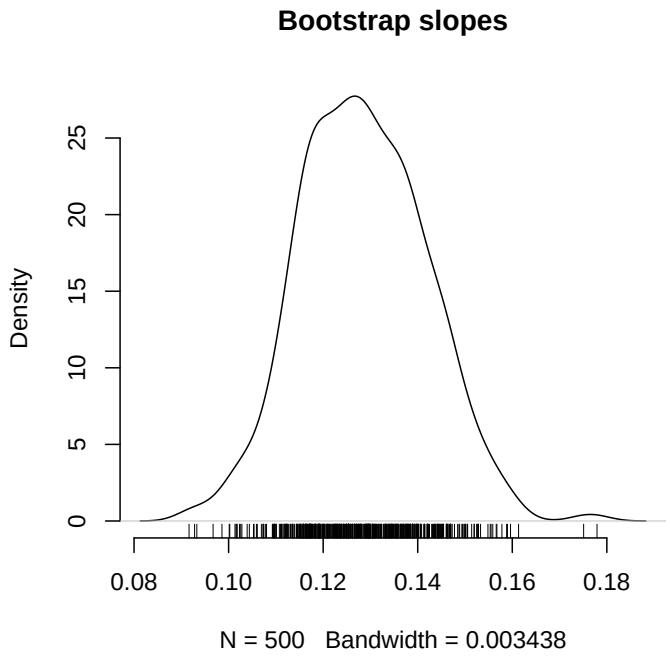

Figure 3.2: Distribution of the bootstrapped slopes for the `cars` dataset

```
bootstrap.slopes = unlist(bootstrap.slopes, use.names = F)
head(bootstrap.slopes)
[1] 0.1022948 0.1339924 0.0985944 0.1412716 0.1249709 0.1345326
```

Let's plot the slope distribution (figure 3.2).

```
# plot the distribution
plot(density(bootstrap.slopes), bty = "n", main = "Bootstrap slopes")
rug(bootstrap.slopes)
```

To determine the 95% confidence interval for the slopes, we can use simple quantiles at 2.5% and 97.5%:

```
quantile(bootstrap.slopes, probs = c(0.025, 0.975))
     2.5%     97.5%
0.1026816 0.1541112
```

The overlay of the bootstrap regression models with the original regression is shown in figure 3.3.

3.3 Exporting variables to each cluster node

In the previous example, each node of the cluster received one input dataset at a time and the function to be applied to it, and was returning the output value. This setup is very simple; sometimes we need to pass more variables to each node. In this section we will see how this can be done using the `snow` library.

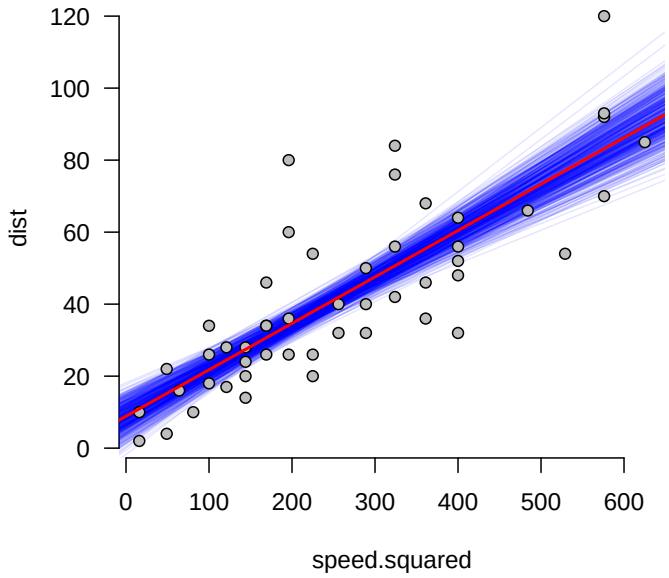

Figure 3.3: Bootstrapped regression models for the `cars` dataset

3.3.1 Flipping coins

Let's consider the following example: we want to simulate the behaviour of a coin which is tossed 100 times. We also want to be able to choose if the coin is fair or not. Here is how to simulate one experiment (100 coin flippings):

```
 coin.sequence = sample(c("H", "T"), size = 100, replace = T)
 table(coin.sequence)
coin.sequence
 H  T
49 51
```

If this case, the coin is fair and the frequency of heads in the sample is 0.49. However, we can specify an expected frequency of heads different from 0.5:

```
 h = 0.25 # head frequency
 coin.sequence = sample(c("H", "T"), size = 100, replace = T,
                        prob = c(h, 1 - h))
 table(coin.sequence)
coin.sequence
 H  T
22 78
```

This time the head frequency is 0.22.

3.3.2 Flipping in parallel

Since we are now able to run independent analyses in parallel, we would like to simulate 100 experiments each comprising 100 flips in order to examine how precise is the head frequency estimate. The preparation for the parallel run is very straightforward: there

is actually no input dataset to store into a list, and only one function to define which will be run in parallel. The output of this function will be stored in a list by `parLapply`. However, to run the function several times in parallel with `parLapply`, we have to use a dummy list as the input.

```
# define a dummy list as an input to parLapply
index = as.list(1:100)
# define the simulation function
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # parLapply, but does not use it.
  sample(c("H", "T"), size = 100, replace = T)
}
```

We can now initialize the cluster and do the run:

```
library(snow)
# define the number of cores
n.cores = 3
# create the cluster with as many cores as needed
cluster = makeSOCKcluster(rep("localhost", n.cores))
# run the simulations
simulations = parLapply(cluster, index, toss.coin)
```

To examine the results, we simply calculate the frequency of heads in each simulations.

```
# apply a function to each simulation
head.frequencies = lapply(simulations,
                          function(x) sum(x == "H") / length(x))
# convert the result to a vector
head.frequencies = unlist(head.frequencies)
head.frequencies[1:10]
 [1] 0.45 0.50 0.49 0.49 0.52 0.49 0.39 0.59 0.55 0.56
# calculate the mean and standard deviation
mean(head.frequencies)
[1] 0.5016
sd(head.frequencies)
[1] 0.04578651
```

In the end, the estimate is quite precise with such a large number of simulations.

3.3.3 Flipping an unfair coin

We can modify slightly the previous function to simulate an unfair coin.

```
# frequency of heads
h = 0.25
# simulation function using h
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # parLapply, but does not use it.
  sample(c("H", "T"), size = 100, replace = T, prob = c(h, 1 - h))
}
# one simulation
simulation = toss.coin()
# observed head frequency
sum(simulation == "H") / length(simulation)
[1] 0.29
```

It works fine. Let's use our parallel cluster again to simulate more experiments.

```
simulations = parLapply(cluster, index, toss.coin)
```

```
Error in checkForRemoteErrors(val) :
  3 nodes produced errors; first error: object 'h' not found
```

This doesn't work so fine. Object `h` is not found. But if we try to call it:

```
h
[1] 0.25
```

The object does exist, but is not found by the cluster. The reason is that we defined `h` in our current workspace, but when the cluster is initialized, each node has its own workspace. Only the list elements it is working on and the function used by `parLapply` are sent to each node, and it does not see the global environment of the calling workspace.

If we want the nodes to know about a variable, we have to explicitly send it to them.

```
# export h to each cluster node
clusterExport(cluster, "h")
# run the simulations
simulations = parLapply(cluster, index, toss.coin)
str(simulations[[1]])
chr [1:100] "T" "T" "T" "H" "T" "T" "T" "T" "T" "T" ...
```

Different strategies can be used to export data to a cluster. Variables can be exported using the `clusterExport` function as shown here, or can also be defined in the body of the function run on each cluster instead of being defined in the global environment. Finally, it is also possible to develop a package to store a project data and function, even though this is more advanced R programming. If it is the case, the package can be loaded inside the body of the function applied by the cluster, and all variables contained within the package will be loaded with the package.

3.4 Parallel computing and random number generation

When performing analyses which rely on random number generation, such as bootstrapping, it is important to ensure that each core will produce an independent stream of random numbers, otherwise the parallel runs will just be exact replicates of each other.

The `snow` library provides functions to initialize independent random number generator (RNG) on each core.

Let's experiment a bit and see if we can constrain our function in order to have exactly identical RNG on each node.

```
# load the library
library(snow)
# set up the cluster
n.cores = 3
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
  set.seed(4) # this initializes the RNG inside the function
  runif(1)
}
```

```
# run the function in parallel
x = as.list(1:9)
a = parLapply(cluster, x, f)
stopCluster(cluster)
unlist(a)
[1] 0.5858003 0.5858003 0.5858003 0.5858003 0.5858003 0.5858003 0
   .5858003
[8] 0.5858003 0.5858003
```

In this case, we force the RNG initialization to the same value, each time, from within the function. Thus, the random numbers are the same between function calls on a single core and also between cores.

What happens if we remove the `set.seed` call? Intuitively, each function call on a single core should generate a new random number, but the random number streams might be the same on the different cores.

```
set.seed(4) # we initialize the seed before the cluster
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
#  set.seed(4) # no seed manipulation here
  runif(1)
}
# run the function in parallel
x = as.list(1:9)
a = parLapply(cluster, x, f)
stopCluster(cluster)
unlist(a)
[1] 0.24599222 0.77893978 0.40450896 0.48764381 0.63350480 0.25539650
   0.08949303
[8] 0.31746022 0.20774874
```

This gives different numbers among the cores. It looks like we don't actually need to initialize different seeds on the cores to have different streams, this might be done automatically when the cluster is set up.

If we run again the same code:

```
set.seed(4) # we initialize the seed before the cluster
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
#  set.seed(4) # no seed manipulation here
  runif(1)
}
# run the function in parallel
x = as.list(1:9)
a = parLapply(cluster, x, f)
stopCluster(cluster)
unlist(a)
[1] 0.02050864 0.78757447 0.44849725 0.20426912 0.86239141 0.89209204
   0.17404899
[8] 0.55943192 0.73525161
```

The numbers are different again, despite the fact that we set up the seed before the cluster initialization. It looks like the seed state outside the cluster has no effect on the RNG on each core.

I didn't dig into it further, but more information about reproducible RNG for parallel computing can be found with the `snowFT` package; see also this post.

But in any case, to ensure the independence of the RNG streams among the cores, it is good practice to use `clusterSetupRNG`:

```
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
  runif(1)
}
# initialize independent RNG
clusterSetupRNG(cluster)
[1] "RNGstream"
# run the function in parallel
x = as.list(1:9)
a = parLapply(cluster, x, f)
stopCluster(cluster)
unlist(a)
[1] 0.1270111 0.3185276 0.3091860 0.7595819 0.9783106 0.6851358 0
   .7285098
[8] 0.9655873 0.9961841
```

By default, `clusterSetupRNG` uses the L'Ecuyer's random number generator (requires the `rlecuyer` package), but another generator is also available.

Interestingly, using `clusterSetupRNG` results in independent RNG even when `set.seed` is used within the function:

```
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
  set.seed(4)
  runif(1)
}
# initialize independent RNG
clusterSetupRNG(cluster)
[1] "RNGstream"
# run the function in parallel
x = as.list(1:9)
a = parLapply(cluster, x, f)
stopCluster(cluster)
unlist(a)
[1] 0.1270111 0.3185276 0.3091860 0.7595819 0.9783106 0.6851358 0
   .7285098
[8] 0.9655873 0.9961841
```

We can also notice that the generated numbers are the same as before. This is because `clusterSetupRNG` used the same default seed, but we can change this:

```
cluster = makeSOCKcluster(rep("localhost", n.cores))
# define a function using random number
f = function(x) {
  runif(1)
}
# initialize independent RNG
clusterSetupRNG(cluster, seed = 1)
[1] "RNGstream"
```

```
 # run the function in parallel
 x = as.list(1:9)
 a = parLapply(cluster, x, f)
 stopCluster(cluster)
 unlist(a)
[1] 0.4570933 0.3185276 0.1073167 0.1804877 0.7473622 0.1301969 0
   .5080424
[8] 0.8141489 0.5439259
```

This is particularly interesting when we want to ensure reproducibility between different runs: we can just use the same seed for `clusterSetupRNG`.

We can also make it different between successive runs by using the time as the seed:

```
 cluster = makeSOCKcluster(rep("localhost", n.cores))
 # define a function using random number
 f = function(x) {
   runif(1)
 }
 # initialize independent RNG
 clusterSetupRNG(cluster, seed = Sys.time())
[1] "RNGstream"
 # run the function in parallel
 x = as.list(1:9)
 a = parLapply(cluster, x, f)
 stopCluster(cluster)
 unlist(a)
[1] 0.622673018 0.318527565 0.433631205 0.236618425 0.295261717 0
   .738553880
[7] 0.004381088 0.788606788 0.616832712
```

Chapter 4

Parallel computing on CSC servers (using the snow library, up to 16 cores)

Parallel computing on a CSC server is very similar to what is done using a local computer with several cores. In this section, we will demonstrate how to use up to 16 cores from one node on the `taito` server.

First, we prepare and test the script on a local computer. Then, we copy it to `taito` and set up the job to be run.

4.1 Preparing the script locally

A script which can run in parallel on a local computer will be able to run in parallel on `taito` without any modification, except the number of cores used. See section 3 for how to write a script that can be run in parallel locally.

Let's use a simple script that flips coins again. Here is the function to simulate a trial (1000000 flippings):

```
# define the simulation function (one trial flips the coin
# 1000000 times)
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # parLapply, but does not use it.
  # In addition, we specify a default value for i so that the
  # function can also be called without argument, outside a lapply
  # context.

  # Simulate the trial
  trial = sample(c("H", "T"), size = 1000000, replace = T)
  # return the proportion of heads
  sum(trial == "H") / length(trial)
}
```

and here is the wrapping for the parallel execution:

```
# we will run 100 coin flipping trials
index = as.list(1:100)
# initialize the cluster
library(snow)
```

```
n.cores = 3 # this will be modified for taito
cluster = makeSOCKcluster(rep("localhost", n.cores))
# run the simulations
simulations = parLapply(cluster, index, toss.coin)
# stop the cluster
stopCluster(cluster)
```

We can have a look at how long it takes to be run locally in serial or in parallel processes, using 3 cores:

```
library(rbenchmark)
# serial function
f.serial = function() {
  lapply(index, toss.coin)
}
# parallel function
f.parallel = function() {
  parLapply(cluster, index, toss.coin)
}
# initialize the cluster
cluster = makeSOCKcluster(rep("localhost", n.cores))
# benchmark
bmk = benchmark(f.serial(), f.parallel(),
                columns = c("test", "replications",
                            "elapsed", "relative"),
                order = "relative",
                replications = 2)
bmk
          test replications elapsed relative
2 f.parallel()            2   5.308    1.000
1   f.serial()            2  21.213    3.996
```

The parallel run is faster than the serial run. The script works and is ready to be copied to `taito`. The main change is that we can now use up to 16 cores for the parallel run (a full node on `taito`).

The script `flipping.coins.R` is:

```
# flipping.coins.R

# flipping coins script for a parallel run on Taito

# trial function
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # parLapply, but does not use it.
  # Simulate the trial
  trial = sample(c("H", "T"), size = 1000000, replace = T)
  # return the proportion of heads
  sum(trial == "H") / length(trial)
}
# initialize the cluster
library(snow)
n.cores = 16 # we can use up to 16 cores on one node with taito
cluster = makeSOCKcluster(rep("localhost", n.cores))
# run
index = as.list(1:100)
# here we use system.time to time the execution of the simulations
# we have to enclose the expression to time between { and }
```

```
# print will force the output to stdout
print(system.time({simulations = parLapply(cluster, index,
                                           toss.coin)}))
# save the results
write.table(unlist(simulations), "flipping.coins.results")
# stop the cluster
stopCluster(cluster)
```

4.2 Setting up the script on Taito

We will run the script on the CSC server Taito. The server address is `taito.csc.fi`. You should have a username and a password to connect to the server. Let's assume your login is `toto` and your password `1234`.

You can connect to a remote terminal on the server by using Putty on Windows or the simple command `ssh toto@taito.csc.fi` from a Mac or GNU/Linux terminal. From the remote terminal, you can start jobs and examine the results of the jobs. To move files to and from Taito, the easiest way is to use a SFTP client.

4.2.1 Copying a file to Taito

You can copy a file to Taito using a SFTP client (e.g., WinSCP, Filezilla). With Filezilla, the host is `taito.csc.fi`, the protocol is `SFTP`, the logon type is `normal` and you can also fill in your login and password.

Once you are connected to Taito with the client, you can create a new folder (`parallel.test`) and copy the script `flipping.coins.R` into it.

4.2.2 Preparing the sbatch file

Each job request on Taito is processed by `sbatch`. To make a request, we have to wrap the call to the R script in a small script which can be used by `sbatch`. More information is available on the CSC website.

Here is the sbatch file we are going to use (`R.snow.sh`):

```
#!/bin/bash -l
#SBATCH -J R_snow
#SBATCH -o output_%J.txt
#SBATCH -e errors_%J.txt
#SBATCH -t 2:00:00
#SBATCH -n 1
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --mem-per-cpu=4000
#SBATCH --mail-type=END
#SBATCH --mail-user=toto@utu.fi

module load R.latest/latest
Rscript --vanilla flipping.coins.R
```

- The first line indicates that it is a bash script (`#!/bin/bash`).
- All the lines starting with `#SBATCH` are comments which are used by `sbatch` to process the job request.

- `-J`: name of the job
- `-o`: file to which output is redirected (%J is a placeholder which will be replaced by the job id during the run).
- `-e`: file to which errors are redirected
- `-t`: time request for the job. The job will be killed if it goes beyond that time limit. The format is `hh:mm:ss`.
- `-n`: number of task run by the job
- `--nodes`: number of node requested. Using `snow` as it is explained in this tutorial, you cannot use more than one node (but it might be possible to do it with `snow` by creating a different type of cluster). If you want to use more than one node, you can read the next section which uses `Rmpi` to run a script with more than one node.
- `--cpus-per-task`: number of cores used by the script. One node on Taito has 16 cores, so `snow` can be run with at most 16 cores (specified in the R script during the cluster initialization).
- `--mem-per-cpu`: each node in Taito has 64 Go of memory. If one uses all 16 cores of a node, then each core can use up to 4000 Mo of memory.
- `--mail-type` and `--mail-user`: useful to get information about when the job starts or ends.
- The last two lines are the actual job and are executed by the shell.

In Taito, users have to load specific modules to have access to different softwares. Details about this system can be found here. Basically, to be able to run R, one should load the R module with:

```
module load R.latest/latest
```

The R script is executed using:

```
Rscript --vanilla flipping.coins.R
```

4.3 Running a job

We now have two files ready in the folder `parallel.test` on Taito: `flipping.coins.R`, which is the R script itself and does the actual analysis, and `R.snow.sh`, which is a bash script with special comment lines which is used to submit the job to the queue.

We can send submit the job to the queue with:

```
sbatch R.snow.sh
```

The job is sent to the queue and will be run when a node is assigned to it. We can follow the status of the job with (replacing `toto` by the real user name):

```
squeue -l -u toto
```

Each job has an id number. To cancel a job, type (replacing `552657` with the real job id number):

```
scancel 552657
```

In our example script, the results are written to `flipping.coins.results`. Any other output is sent to `output_xxx` (e.g., the execution time returned by `system.time`) and errors are sent to `errors_xxx`, where xxx is the job id.

4.4 Comparison with a serial job

The script can be submitted in a serial form to compare execution times (saved in `flipping.coins.serial.R`):

```
# flipping.coins.serial.R

# flipping coins script for a serial run on Taito

# trial function
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # lapply, but does not use it.
  # Simulate the trial
  trial = sample(c("H", "T"), size = 1000000, replace = T)
  # return the proportion of heads
  sum(trial == "H") / length(trial)
}
# run
index = as.list(1:100)
# here we use system.time to time the execution of the simulations
# we have to enclose the expression to time between { and }
# print will force the output to stdout
print(system.time({simulations = lapply(index, toss.coin)}))
# save the results
write.table(unlist(simulations), "flipping.coins.results.serial")
```

The sbatch file is (`R.serial.sh`):

```
#!/bin/bash -l
#SBATCH -J R_serial
#SBATCH -o output_%J.txt
#SBATCH -e errors_%J.txt
#SBATCH -t 2:00:00
#SBATCH -n 1
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=4000
#SBATCH --mail-type=END
#SBATCH --mail-user=toto@utu.fi

module load R.latest/latest
Rscript --vanilla flipping.coins.serial.R
```

During my tests, the execution time was 0.656s with a parallel process and 6.242s with a serial job. The improvement is about 10 fold, which is less than expected based on the number of cores used (16), but in this case the analysis is a short one and the overhead

of the distribution of the analyses on the cores might not be negligible compared to the analysis time itself. However, analyses that need to be run in parallel are typically long ones for which this overhead is likely to be very small compared to the analysis run time.

Chapter 5

Parallel computing on CSC servers, using more than 16 cores (`Rmpi` library)

The use of the `snow` library described in this tutorial does not allow to use more than one node for a parallel run, and thus limits the number of cores that can be used on `Taito` to 16. It might be possible to go beyond this limit using `snow`, but I didn't find how to use it for this purpose. Instead, I used the `Rmpi` library.

The `Rmpi` library can be used in the same way as `snow`:

- Store the input data as a list.
- Define a function to perform the analysis. This function will be applied to each element of the list, independently.
- Run the analysis using a parallelized version of `lapply`.

The differences are extremely small and a script which is running successfully with `lapply` or `parLapply` with `snow` can be modified to run with `Rmpi` in a few minutes. The advantage is that several nodes can then be requested on `Taito`, enabling to use e.g. 32 or 64 cores at a time.

5.1 Preparing the script locally

We use the same coin-flipping script as in the previous section with minor modifications (`flipping.coins.Rmpi.R`).

```
# flipping.coins.Rmpi.R

# flipping coins script for a parallel run on Taito

# trial function
toss.coin = function(i = 0) {
  # This function takes one dummy argument in order to be callable by
  # parLapply, but does not use it.
  # Simulate the trial
  trial = sample(c("H", "T"), size = 1000000, replace = T)
  # return the proportion of heads
  sum(trial == "H") / length(trial)
```

```
}
# initialize the cluster
library(Rmpi)
# run
index = as.list(1:100)
# here we use system.time to time the execution of the simulations
# we have to enclose the expression to time between { and }
# print will force the output to stdout
print(system.time({simulations = mpi.parLapply(index, toss.coin)}))
# save the results
write.table(unlist(simulations), "flipping.coins.results")
# stop the cluster
mpi.close.Rslaves()
mpi.quit()
```

The differences are:

- We use `library(Rmpi)` instead of `library(snow)`.
- We don't have to initialize the cluster and to specify the number of cores: `Rmpi` will use all the available cores when the library is loaded. The number of cores is thus entirely determined in the `sbatch` file for the job submission.
- We use `mpi.parLapply` instead of `parLapply` and we don't have to specify the cluster in the arguments.
- We stop the cluster using `mpi.close.Rslaves()` and `mpi.quit()`.

5.2 Preparing the sbatch file

Here is the sbatch file we use with `Rmpi` (`R.Rmpi.sh`):

```
#!/bin/bash -l
#SBATCH -J Rmpi
#SBATCH -o output_%J.txt
#SBATCH -e errors_%J.txt
#SBATCH -t 2:00:00
#SBATCH -n 64
#SBATCH -p parallel
#SBATCH --mem-per-cpu=4000
#SBATCH --mail-type=ALL
#SBATCH --mail-user=toto@utu.fi

module load R.latest/latest
srun -u -n 64 Rmpi --no-save < flipping.coins.Rmpi.R
```

The differences with the previous `sbatch` file are:

- We use the `-n` option to ask for 64 cores.
- We specify that the job has to be run on the parallel partition with `-p parallel`
- We don't have to specify the number of nodes nor the number of cpus per task.
- Here we also ask for mail for `ALL` actions (i.e. start and end) since it sometimes takes time to start the job, and it is good to know that it actually started.

Note: We have to be careful and specify the same number of requested cores between the `-n` option and the `srun` command (last line).

5.3 Running a job

Our project folder on Taito now contains two files:

- The R script `flipping.coins.Rmpi.R`
- The sbatch file `R.Rmpi.sh`

We can submit the job as usual by typing:

```
sbatch R.Rmpi.sh
```

5.4 Random number generation with Rmpi

To illustrate how to generate random numbers with `Rmpi`, here is a small R script generating random numbers and writing them to a file:

```
# RNG.Rmpi.R

# generate random numbers and write them to a file

# function to generate the random numbers
generate.random = function(i = 0) {
  # again, i is just a dummy parameter so that the function can be
  # called by lapply or mpi.parLapply
  rnorm(1)
}
# initialize the cluster
library(Rmpi)
mpi.setup.rngstream() # initialize the RNG on each core
# run
index = as.list(1:128)
results = mpi.parLapply(index, generate.random)
# save the results
write.table(unlist(results), "random.numbers.Rmpi")
# stop the cluster
mpi.close.Rslaves()
mpi.quit()
```

Here is the corresponding sbatch file:

```
#!/bin/bash -l
#SBATCH -J Rmpi
#SBATCH -o output_%J.txt
#SBATCH -e errors_%J.txt
#SBATCH -t 0:05:00
#SBATCH -n 64
#SBATCH -p parallel
#SBATCH --mem-per-cpu=4000
#SBATCH --mail-type=ALL
#SBATCH --mail-user=toto@utu.fi

module load R.latest/latest
srun -u -n 64 Rmpi --no-save < RNG.Rmpi.R
```

Using this script generates 128 different numbers, which means that each core generated different random numbers.

Running this script twice will result in a different set of 128 numbers. This is because `mpi.setup.rngstream()` is called without any argument. To set up reproducible RNG between different runs, we can use:

```
mpi.setup.rngstream(iseed = 4)
```

This will ensure that the same set of 128 random numbers is generated for each run. As is explained in `mpi.setup.rngstream` help, `iseed` can take any integer, and if set to `NULL` (the default value) non reproducible random numbers will be generated.