Parallel computations in R

Introduction

In this exercise we will have a look at how to run parallel computations in R, i.e. how to run multiple computations at different cores of your CPU at the same time. Doing your computations in parallel can be worthwhile if you do many similar time-consuming computations. For example, if you want to evaluate the behaviour of your complex models in a large number of simulations, or if you want to run a complex model for a large number of datasets, e.g. for different stations, areas, or variables.

Make sure you have installed and loaded the following packages:

```r
library(tidyverse)
library(mgcv)
library(furrr)
```

In the exercise "Chlorophyll: group_by - filter - summarize - map" we saw how `map` could be used to apply the same function to several different datasets. The `furr` packages lets us run such computations in parallel, using a new function called `future_map`, which is used analogously to `map`. After loading the package, we run `plan` to set the number of parallel workers to use, i.e. how many computations to perform at a time. For instance, if we want to use 3 cores in parallel:

```r
plan(multisession, workers = 3)
```

Simulate data

First, we start by creating a function that simulates a sine-function and adds a random error term with a standard deviation of 0.1. We create 1,000 such datasets using `map`, and then merge them to a data frame called `df`, where the variable `subs` indicates which dataset each row belongs to.

```r
simulate_sin <- function(a){
  x = 1:100
  y = sin(seq(0, 5*pi, length.out = 100)) + rnorm(n=100, mean = 0, sd=0.1)
  return(data.frame(x,y))
}

alldata <- map(1:1000, simulate_sin)
df <- data.frame(do.call(rbind, alldata)) %>%
       mutate(subs = rep(1:1000, each = 100))
```

Fitting models to subsets in parallel

Now, we fit a GAM (generalized additive model) to each dataset. We use `gam` to fit the model, and `map` to apply this function to each subset of the data. `gam` is wrapped in `possibly`, which means that R won't stop if there is an error in one of the iterations. We use the `system.time` function to check how long this takes to run (expect it to take a minute or so, depending on how fast your computer is):

```r
system.time(
df1 <- df %>%
  group_by(subs) %>%
  split(.$subs) %>%
  map(possibly(~gam(formula= y~s(x), data=.x, method="REML", select=T),
               otherwise = NA_real_)))
```

The time is presented using three measurements: user time is the CPU time charged for the execution of user instructions of the calling process, system time is the CPU time charged for execution by the system on behalf of the calling process, and elapsed time is how long the code chunk took to run.

To run the same computations in parallel, we replace `map` by `future_map`:

```
system.time(
df2 <- df %>%
  group_by(subs) %>%
  split(.$subs) %>%
  future_map(possibly(~gam(formula= y~s(x), data=.x, method="REML", select=T),
                      otherwise = NA_real_)))
```

As you can see, the computations run faster in parallel. If you have a CPU with a high number of cores, you can increase `workers` in `plan` even more, and increase the speed further. To see how many cores are available on your system, run:

```
availableCores()
```

It is generally a good idea not to use all your cores for the parallel computations - at least one will always be needed for other tasks (running RStudio, your operating system, other applications, etc.).

Exercise Use `map2`, `future_map2` and `predict` to make predictions for each subset in `df`, using the fitted model corresponding to each dataset.

Running `for` loops in parallel

`for` loops are used for iteration, like in this example:

```
# Non-parallel for loop:
loop_output <- rep(NA, 9)
for(i in 1:9)
{
      loop_output[i] <- i^2
}
```

The `foreach` and `doParallel` packages allow you to run `for` loops in parallel using `foreach`. Unlike base R's `for` loop,s the output from each iteration is stored in a list. Here is an example:

```
# Parallel version:
library(foreach)
library(doParallel)
registerDoParallel(3) # Use 3 cores

loop_output <- foreach(i = 1:9) %dopar%
{
      i^2
}

loop_output # The output is stored as a list
unlist(loop_output) # Turn it into a vector instead
```

Note that the list needn't be ordered by the iteration index - it is possible that the computation for `i=5` finished before the computation for `i=4`, in which case the result for `i=5` will appear in the list before the result for `i=4`. In this case, and in other cases where the output from each iteration is a vector rather than a scalar, we can collect the output in a matrix using `matrix`:

```r
loop_output <- foreach(i = 1:9) %dopar%
{
      c(i, i^2) # Save both i and i^2
}

loop_output
# Convert to a matrix with 9 rows and 2 columns:
matrix(unlist(loop_output), 9, 2, byrow = TRUE)
```

Further examples on parallelisation can be found in Chapter 6 of *Modern Statistics with R*.

Solutions to exercises

First, we fit the models as before:

```r
models <- df %>%
  group_by(subs) %>%
  split(.$subs) %>%
  map(possibly(~gam(formula= y~s(x), data=.x, method="REML", select=T),
               otherwise = NA_real_))
```

Next, we split the dataset into subsets, and use `map2` to make predictions for each subset:

```r
split_data <- df %>%
  group_by(subs) %>%
  split(.$subs)

predictions <- map2(models, split_data, predict)
```

The parallelized version of this is:

```r
models <- df %>%
  group_by(subs) %>%
  split(.$subs) %>%
  future_map(possibly(~gam(formula= y~s(x), data=.x, method="REML", select=T),
                      otherwise = NA_real_))

split_data <- df %>%
  group_by(subs) %>%
  split(.$subs)

predictions <- future_map2(models, split_data, predict)
```