Solutions

Jumping Rivers

Practical 1

Practical 1

1. Reproduce the timing results in chapter 2 using the `mark()` function from the `bench` package. Remember to load the package using

   ```
   library("bench")
   ```

2. **Case study** In this example, we are going to investigate loading a large data frame. First, we'll generate a large matrix of random numbers and save it as a csv file:[1]

   ```
   N = 1e+05
   m = as.data.frame(matrix(runif(N), ncol = 1000))
   write.csv(m, file = "example.csv", row.names = FALSE)
   ```

   We can read the file the back in again using `read.csv`

   ```
   dd = read.csv("example.csv")
   ```

   To get a baseline result, time the `read.csv` function call above.

   ```
   system.time(read.csv("example.csv"))
   ```

   ```
   ##    user  system elapsed
   ##   0.127   0.000   0.127
   ```

   We will now look ways of speeding up this step.

   - Explicitly define the classes of each column using `colClasses` in `read.csv`, for example, if we have 1000 columns that all have data type numeric, then:

     ```
     read.csv(file="example.csv",
              colClasses=rep("numeric", 1000))
     ```

   - Use the `saveRDS` and `readRDS` functions:

     ```
     saveRDS(m, file = "example.RData")
     readRDS(file = "example.RData")
     ```

   - Install the `readr` package via

     ```
     install.packages("readr")
     ```

     Then load the package in the usual way

     ```
     library("readr")
     ```

[1] If setting `N=1e6` is too large for your machine, reduce it at bit. For example, `N=50,000`.

> This package contains the function `read_csv`; a replacement function for `read.cvs`. Is this new function much better than the default?

Which of the above give the biggest speed-ups? Are there any downsides to using these techniques? Do your results depend on the number of columns or the number of rows?

```
## 1. Using RData files is the fastest -
## although you have to read the data in first.
## Set colClasses also produces an good
## speed-up.

## 2. Setting colClasses R is no longer
## checking your data types.  If your data is
## changing - for example it's coming from the
## web or a database, this may be problem.

## 3. The results do depend on the number of
## columns, as this code demonstrates
N = c(1, 10, 100, 1000, 1000, 10000)
l = numeric(5)
for (i in seq_along(N)) {
    m = as.data.frame(matrix(runif(N[6]), ncol = N[i]))
    write.csv(m, file = "example.csv", row.names = FALSE)
    cc = rep("numeric", N[i])
    l[i] = system.time(read.csv("example.csv",
        colClasses = cc))[3]
}
l

## Notice that when we have a large number of
## columns, we get a slow down in reading in
## data set (even though we have specified the
## column classes). The reason for this slow
## down is that we are creating a data frame
## and each column has to be initialised with a
## particular class.
```

Practical 2

1. In this question, we'll compare matrices and data frames. Suppose we have a matrix, `d_m`

   ```
   ## For fast computers d_m = matrix(1:1000000,
   ## ncol=1000) Slower computers
   d_m = matrix(1:10000, ncol = 100)
   dim(d_m)
   ```

   ```
   ## [1] 100 100
   ```

   and a data frame `d_df`:

   ```
   d_df = as.data.frame(d_m)
   colnames(d_df) = paste0("c", 1:ncol(d_df))
   ```

   - Using the following code, calculate the relative differences between selecting the first column/row of a data frame and matrix.

     ```
     mark(d_m[1,], d_df[1,], d_m[,1], d_df[,1],
          relative = TRUE,
          check = FALSE)
     ```

     Can you explain the result? Try varying the number of replications.

     ```
     ## Two things are going on here 1. The very
     ## large difference when selecting columns and
     ## rows (in data frames) is because the data is
     ## stored in column major-order. Although the
     ## matrix is also stored in column major-order,
     ## because everything is the same type, we can
     ## efficiently select values.

     ## 2. Matrices are also more memory efficient:
     m = matrix(runif(10000), ncol = 10000)
     d = data.frame(m)
     object.size(m)
     ```

     ```
     ## 80200 bytes
     ```

     ```
     object.size(d)
     ```

     ```
     ## 1120568 bytes
     ```

     – When selecting columns in a data frame, there are a few different methods. For example,

```
d_df$c10
d_df[, 10]
d_df[, "c10"]
d_df[, colnames(d_df) == "c10"]
```

Compare these four methods.

```
mark(d_df$c10, d_df[, 10], d_df[, "c10"], d_df[,
    colnames(d_df) == "c10"], iterations = 10000,
    relative = TRUE)

m = matrix(1:1e+08, ncol = 10000)
dim(m)

mark(m[, 1], m[1, ], iterations = 10000, relative = TRUE)
```

2. Consider the following piece of code:

```
a = NULL
for(i in 1:n)
  a = c(a, 2 * pi * sin(i))
```

This code calculates the values:

$$2\pi \sin(1), 2\pi \sin(2), 2\pi sin(3), \ldots, 2\pi sin(n)$$

and stores them in a vector. Two obvious ways of speeding up this code are:

- Pre-allocate the vector `a` for storing your results.
- Remove $2 \times \pi$ from the loop, i.e. at the end of the loop have the statement: `2*pi*a`. Try the above techniques for speeding up the loop. Vary n and plot your results.

3. R is an interpreted language; this means that the interpreter executes the program source code directly, statement by statement. Therefore, every function call takes time.[2] Consider these three examples:

[2] This example is for illustrative proposes. Please don't start worrying about comments and brackets.

```
n = 1e6
## Example 1
I = 0
for(i in 1:n) {
  10
  I = I + 1
}
## Example 2
I = 0
```

```r
for(i in 1:n){
  ((((((((((10))))))))))
  I = I + 1
}
## Example 3
I = 0
for(i in 1:n){
  ##This is a comment
  ##But it is still parsed
  ##So takes time
  ##But not a lot
  ##So don't worry!
  10
  I = I + 1
}
```

Using the `mark()` function, time these three examples.

Practical 3: parallel programming

1. To begin, load the `parallel` package and determine how many cores you have

```
library("parallel")
detectCores()
```

2. Run the parallel `apply` example in the notes.

- On your machine, what value of `N` do you need to use to make the parallel code run quicker than the standard serial version?
- When I ran the benchmarks, I didn't include the `makeCluster` and `stopCluster` functions calls. Include these calls in your timings. How does this affect your benchmarks?

3. Run the dice game Monte-Carlo example in the notes. Vary the parameter `M`.[3]

[3] Try setting `M=50` and varying `N`.

Solutions

Solutions are contained within this package:

```
library("jrEfficient")
vignette("solutions1", package = "jrEfficient")
```