Benchmarking R Codes

Introduction

Benchmaking is the process of testing the performance of specific operations *repeatedly*. Benchmarking R codes allows us to compare the speed at which different sets of codes/functions run and will subsequently inform us of the most efficient set of code/function to use. Previously in *Efficient R Codes*, we looked into different strategies for writing efficient R codes, namely avoid growing vectors in `for` loops, utilizing `apply` family as an alternative to `for` loops, exploiting parallel computing, and using vectorized codes. We used `system.time()` function to determine the time required to execute each strategy. Nevertheless, this approach has several limitations:

- Allows evaluation of only a single set of codes at any one time.
- Returns slightly different results when tested on the same set of codes (due to stochasticity).
- Unable to detect small measurements, e.g. microseconds.

The `microbenchmark()` function from its eponymous package can be used to address these limitations of `system.time()`:

- Allows evaluation and comparison of multiple sets of codes.
- Computes the average runtime by evaluating the same set of codes multiple times. This mitigates stochastic effect in the evaluated runtime.
- Able to detect small differences in runtime between different sets of codes, i.e. up to nanoseconds.

```
install.packages("microbenchmark")
```

Dataset

In this tutorial, we will be using the GENCODE gene transfer file (GTF) file. A GTF file contains the comprehensive gene, transcript, and exon annotations for a give species. The latest version of a GTF file can be retrieved from the GENCODE repository (https://www.gencodegenes.org/). Here, we will using the human GTF file version 31. The data frame consists of 9 columns and a brief explanation of each of these columns as follows:

```
# Load package
library(data.table)

# Read file
df <- fread("Datasets/gencode.v31.annotation.gtf", sep="\t",
            header=FALSE, stringsAsFactors=FALSE)

# Subset required feature
df <- df[which(df$V3=="gene"), ]

# Check dimensions
dim(df)
```

```
## [1] 60603     9
```

Benchmarking

Here, we will benchmark the different sets of R codes used previously in *Efficient R Codes*, namely avoid growing vectors in `for` loops, utilizing `apply` family as an alternatives to `for` loops, exploiting parallel computing, and using vectorized codes against a `for` loop that grows its vector. The `times` option specifies the no. of times to evaluate each set of codes. The default is 100 times, but we'll do just 5 times in the interest of time.

```r
# Load package
library(microbenchmark)
library(parallel)

# Create cl object for parallel computing
cl <- makeCluster(4)

# Perform benchmarking
microbenchmark(

  # for loop that grows its vector
  "loop.grow.vector"={

    vec <- NULL
    for(i in 1:nrow(df)) {
        df.small <- df[i, ]
        v9.split <- strsplit(df.small$V9, split=";")
        attr <- sapply(v9.split, function(x) {x[3]})
        vec <- c(vec, attr)
    }

  },

  # Avoid growing vector
  "loop.dont.grow.vector"={

    vec <- numeric(nrow(df))
    for(i in 1:nrow(df)) {
      df.small <- df[i, ]
      v9.split <- strsplit(df.small$V9, split=";")
      attr <- sapply(v9.split, function(x) {x[3]})
      vec[i] <- attr

    }

  },

  # Using the apply function
  "apply.function"={

    retrieve_gene_name <- function(x) {
      y <- strsplit(x[9], split=";")
      sapply(y, function(z) {z[3]})
    }
    attr <- apply(df, 1, retrieve_gene_name)

  },

  # Parallel computing
  "apply.parallel.computing"={

    retrieve_gene_name <- function(x) {
      y <- strsplit(x[9], split=";")
```

```
      sapply(y, function(z) {z[3]})
    }
    attr <- parApply(cl, df, 1, retrieve_gene_name)

  },

  # Vectorized code
  "vectorized.code"={

   attr <- sapply(strsplit(df$V9, split=";"), function(x) {x[3]})

  },

  times=5

)
```

```
## Unit: milliseconds
##                       expr        min         lq       mean     median
##          loop.grow.vector 31705.7797 31974.6465 37562.8996 38783.0787
##     loop.dont.grow.vector  8579.0193  9328.9566  9778.0494  9344.2952
##            apply.function  2144.3648  2391.1555  2483.5566  2521.9484
##  apply.parallel.computing   998.8424  1043.1820  1067.8027  1057.9070
##           vectorized.code   684.2595   685.8866   713.0767   691.7906
##          uq        max neval
##  39656.4454 45694.5474     5
##   9799.1738 11838.8022     5
##   2630.3802  2729.9342     5
##   1088.7017  1150.3801     5
##    732.3534   771.0934     5
```

- Clearly using the `apply()` function and vectorized code are more efficient among all the sets of codes tested.
- For the `for` loop approach, initialising the vector with final length and substituting the values by subscripting gives superior performance compared to growing the vector.
- For the `apply()` approach, parallel computing increases the function's efficiency.

Reference

Gillespie, C. and Lovelace, R. 2017. *Efficient R Programming*. O'Reilly Media.