Efficient R Codes

Introduction

All roads lead to Rome. In R programming, there are many ways to write codes to achieve the same objective, be it in data cleaning or data analysis. Nevertheless, the faster the codes run, the more efficient they are. This is particularly relevant in data science or in the era of next-generation sequencing where large data structures and highly repeatitive tasks are the norm. Writing efficient R codes can ultimately save you a significant amount of time as we will see in the following examples. Strategies for writing efficient R codes are (but not limited to):

- Never grow objects in `for` loops.
- Utilizing apply family as an alternative to `for` loops.
- Parallel computing
- Using vectorized codes
- Reading files efficiently

The packages needed in this tutorial are **data.table** and **parallel**. You should have them installed if you have not already done so.

```
install.packages(data.table)
install.packages(parallel)
```

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:

Column	Content	Value
1	Chromosome name	1, 2, 3
2	Annotation source	ENSEMBLE, HAVANA
3	Feature type	gene, transcript, exon
4	Genomic start location	interger
5	Genomic end location	interger
6	Score (not used)	.
7	Genomic strand	+, -
8	Genomic phase (for CDS features)	., 0, 1, 2
9	Attributes	gene_id, transcript_id, gene_type, gene_status, gene_name, transcript_ty

```
# 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
```

```r
dim(df)
```

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

Never grow objects

This is relevant in writing `for` loops where we would first initialise an object (vector, list, matrix etc.) and subsequently fill in the object after each loop. Here, we would like to retrieve the gene names from each row (gene) of the data frame.

- Method 1: Initialise an **empty** vector and gradually increase its length.

```r
system.time({

  # Initialise vector
  vec <- NULL

  # Retrieve gene_name for each gene
  for(i in 1:nrow(df)) {

    # Subset data frame
    df.small <- df[i, ]

    # Split attribute column
    v9.split <- strsplit(df.small$V9, split=";")

    # Retrieve selected attribute
    attr <- sapply(v9.split, function(x) {x[3]})

    # Grow vector
    vec <- c(vec, attr)

  }

})
```

```
##    user  system elapsed
##  73.757   3.565  40.011
```

```r
head(vec)
```

```
## [1] " gene_name \"DDX11L1\""     " gene_name \"WASH7P\""
## [3] " gene_name \"MIR6859-1\""   " gene_name \"MIR1302-2HG\""
## [5] " gene_name \"MIR1302-2\""   " gene_name \"FAM138A\""
```

- Method 2: Create of a vector of **final length** and change the values in the vector by **subscripting**.

```r
system.time({

  # Define final length of vector
  vec <- numeric(nrow(df))

  # Retrieve gene_name for each gene
  for(i in 1:nrow(df)) {

    # Subset data frame
    df.small <- df[i, ]
```

```
    # Split attribute column
    v9.split <- strsplit(df.small$V9, split=";")

    # Retrieve selected attribute
    attr <- sapply(v9.split, function(x) {x[3]})

    # Replace value in vector by subscripting
    vec[i] <- attr

  }

})
```

```
##    user  system elapsed
##  17.061   0.174   8.894
```

The apply family

The apply family can be a faster alternative to `for` loops. Commonly used `apply` family functions are `apply()`, `sapply()`, and `lapply()`.

- `apply()` applies a function to each row or column of a data frame or matrix and returns a vector.
- `lapply()` is similar to `apply()` but it applies a function to a vector or list and returns a list.
- `sapply()` is similar to `lapply()` and it returns either a vector, list or data frame depending on the context.

Other members of the apply family include `eapply()`, `mapply()`, `rapply()`, and `tapply()`.

```
system.time({

  # Create function to retrieve gene_name
  retrieve_gene_name <- function(x) {

    # Select 9th column and split attributes
    y <- strsplit(x[9], split=";")

    # Retrieve gene_name
    sapply(y, function(z) {z[3]})

  }

  # Apply function
  attr <- apply(df, 1, retrieve_gene_name)

})
```

```
##    user  system elapsed
##   2.364   0.015   2.372
```

Parallel computing

It is possible to combine parallel computing with the `apply` family. This is one of the many advantages of using the `apply` family over `for` loops. We will be using the **parallel** package here. You may first determine the no. of cores on your local machine.

```r
# Load package
library(parallel)

detectCores()
```

```
## [1] 4
```

Using the `apply` family function in parallel computing is straightforward with only two main pointers to take note of:

- The parallel version of the `apply` functions have the prefix par, i.e. `parApply()`, `parLapply()`, and `parSapply()`.
- The additional option `cl` is needed and it comes first in the parallel-apply functions. This option indicates the no. of processors to use and it takes an object created by `makeCluster()`.

```r
# Create cl object
cl <- makeCluster(4)

system.time({

  # Create function to retrieve gene_name
  retrieve_gene_name <- function(x) {

    # Select 9th column and split attributes
    y <- strsplit(x[9], split=";")

    # Retrieve gene_name
    sapply(y, function(z) {z[3]})

  }

  # Apply function
  attr <- parApply(cl, df, 1, retrieve_gene_name)

})
```

```
##    user  system elapsed
##   0.423   0.025   1.234
```

Vectorize code

For all intents and purposes of this tutorial, running a `for` loop or using the `apply` family wasn't necessarily, i.e. we didn't need to go through each and every row to retrieve the gene names. We could succinctly retrieve the gene names using a vectorized code. *Vertorized* refers to the function's input and/or output naturally work with vectors and thus reducing the no. of function calls required.

```r
system.time({

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

})
```

```
##    user  system elapsed
##   0.678   0.004   0.701
```

Reading files efficiently

Typically, we would use `read.table()` for reading in tab-delimited files. If you have an eagle eye, you would have noticed that we did not this function to read in the GTF file. This particular file is more than 1GB in size! Instead, here we used `fread()` from the **data.table** package to read in the file. Another comparatively fast read-in function `read_delim()` comes from the **readr** package, but this function requires more explicit specification of its arguments and it determines the column's class by sampling only the first 1,000 rows. Let's compare the time taken to read in the GTF files using the first two approaches.

```
# read.table()
system.time({

  df <- read.table("Datasets/gencode.v31.annotation.gtf", sep="\t",
                   header=FALSE, stringsAsFactors=FALSE)

})
```

```
##    user  system elapsed
##  28.827   0.592  29.919
```

```
# fread()
system.time({

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

})
```

```
##    user  system elapsed
##  10.513   0.384   5.607
```

Reference

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