How to speed up your R computation by vectorization and parallel programming

Lecture 1

1. Introduction
2. Knowing R data objects/structures and functions
3. Some examples
4. Monte Carlo simulation and apply functions

Introduction

- The main purpose of this workshop is to introduce you how to speed up your R computation by vectorization, interfacing and coding in C/C++ level, and parallel programming
- The first improvement of your R codes, also easiest to implement, is to vectorize the codes as much as possible (Lecture 1)

- For some computations such as recursive looping in time series, MCMC, etc, porting R codes into C/C++ level can speed up dramatically (Lecture 2)
- With the above improvements, if the computation time still requires weeks or months to finish, you need to work with parallel programming:
  a big computation job must be broken down into many small parts with which they can be run concurrently
- Possible reasons for large computation: data are too big or/and models/methods are too complicated
- Do you have any other reasons?
- Lecture 3 will introduce you to parallel programming environment, terminologies, hardware and software, and how to run embarrassing parallel on a single PC with multi-cores
- Lecture 4 will introduce you to MPI and some basic MPI operations and how to use R package Rmpi to run R codes in a supercomputer such as SHARCNET
- Is parallel programming hard, and if so, what can you do about it?
  - ⋆ Added complexity: Computation must be broken down into many small parts
  - ⋆ Parallel programming is error-prone which makes debugging much hard

- ⋆ Too little knowledge of new, innovative parallel cluster systems
  - ⋆ ...
- Facts: parallel programming was hard when it was implemented at C or Fortran level
- R, an interpreted language, shows its advantage in parallel programming
  - ⋆ R has advanced data structure and management
  - ⋆ It is relatively easy to move data among computation jobs in R
  - ⋆ No compiling is required to run parallel computing in R

Knowing R data objects/structures and functions

- Vectors
  - ⋆ Three basic vectors: integer, double, and character vectors
  - ⋆ They are the simplest data objects
  - ⋆ Vectorization: Treat vectors as the smallest objects and carry out all computations as though they are like single numbers (without any explicit looping)
  - ⋆ A simple example:

```
> y=sin(x)
```

  - Looping way

    ```
    > y=double(length(x))
    > for (in in 1:length(x)) y[i]=sin(x[i])
    ```

- Other data objects
  - Matrix and data.frame: ordered set of vectors with equal length
  - List: a collection of objects; useful for outputs rather than inputs
- Logical expressions of vectors
  - ==,!=,<, >, <=, >=,|, &, all, any
  - TRUE (=1), FALSE (=0)
  - Extremely useful to work with subsets of vectors (matrices, data.frames, etc)
  - An example: x — a vector of p values, calculate rejection rate of 0.05 level

    ```
    > mean(x > 0.05)
    ```

  - Another example: remove all missing values in a vector x

    ```
    > x=x[!is.na(x)]
    ```

- Some useful functions with vectors as inputs
  - Creating vectors: c, rep, :, seq
  - Find attributes of a vector: length, mode, class, names
  - Other functions:
    - as.integer, as.double, is.integer, is.double, ...
    - mean, sum, abs, rank, order, ...
- Structure of R functions
  - new_function_name = function(arguments){

    + Function body (R expressions)

    + Return values, Side Effects

    }
  - It is crucial to prepare arguments=inputs well in advance and put all required data objects into arguments
  - Avoiding accessing global data objects in function body by all means
    - Global objects: Any objects that are created outside a function are global objects related to this function
    - Local objects: Any objects created in a function are temporary and will be lost after exiting the function

- Programming Style
  - ⋆ Modularize your codes
  - ⋆ Comment your codes
  - ⋆ Document your codes
  - ⋆ Use proper indent
  - ⋆ Use existing functions
  - ⋆ Use parentheses to make grouping explicitly
  - ⋆ Avoid unnecessary looping

Some examples

- Rejection method
  - ⋆ density of interest: $f(x), a \leq x \leq b$
  - ⋆ a known function: $M(x) \geq f(x), a \leq x \leq b$
  - ⋆ algorithm: let m(x)=M(x)/(integral of M over [a,b])
  - ⋆ step1: Generate T with the density function $m(x)$
  - ⋆ step2: Generate U of Unif[0, 1]. If $M(T) * U \leq f(T)$ then $X = T$ else go step1

- R codes
  - Assume: a, b — finite, m(x)— unif[a,b]
  - Simulate one observation
  - Create a vector
  - Vectorized method

- Simulate mixture distributions
  - Mixture distribution:

$$M(x) = \alpha_1 F_1(x) + \cdots + \alpha_k F_k(x),$$

    where $\alpha_i > 0$, $\alpha_1 + \cdots + \alpha_k = 1$, and $F_i(x)$ is a CDF for $i = 1, \ldots, k$
  - Algorithm:
    1. Generate U following Binomial($\alpha_1, \ldots, \alpha_k$) or generate $U = i$ with probability $\alpha_i, i = 1, \ldots, k$
    2. If $U = i$, generate M according to the distribution $F_i(x)$
  - Simulate a mixture of normals (normal with outliers)

```
95%normal(mu, sigma^2)+5%normal(mu, (k*sigma)^2)
```

- ⋆ Simulate a mixture of 4 distributions
  - ∗ A loop will work but is not efficient
  - ∗ Using replicate results no much improvement
  - ∗ Vectorized way is much more efficient

- Find a MLE in parameter estimation
  - ⋆ density: f(x,theta)
  - ⋆ data x
  - ⋆ log likelihood l(theta,x)=sum of log f(x, theta)
  - ⋆ vectorization: f(x, theta) can take vector x rather than f(x[i],theta)

- Work with ecdf function
  - ⋆ ecdf takes a vector as input and outputs as a function
  - ⋆ ecdf's output can take a vector as input
  - ⋆ Vectorization can be done

- Summary: Vectorization is not difficult to implement as long as computation can be carried in "parallel" way

Monte Carlo simulation and apply functions

- A typical simulation procedures
  - ⋆ DGP (data generating process): need to produce data x
  - ⋆ Modeling: a specific model under consideration
  - ⋆ Estimation: use model data to estimate θ (pretend it is unknown)
  - ⋆ Start the loop: Carry out the real simulation
    - ∗ Need to choose sample sizes
    - ∗ Need to choose simulation sizes
    - ∗ Proper use of replicate function or apply function
  - ⋆ Analyze simulation results
- Before starting the loop, it is very important to implement one simulation as efficient as possible
- Use R's apply functions: apply, lapply, sapply, replicate, etc to start the loop
  - ⋆ replicate(10000, one.simulation(n, theta))
  - ⋆ sapply(rep(n, 10000), one.simulation, theta=theta)
- If simulation takes much long time, you may try to use parallel versions of apply functions