Biostatistics 615 - Statistical Computing

Lecture 12 Advanced R – Performance

Jian Kang

Nov 3, 2015

Microbenchmarking

Accurate Timing Functions

- A microbenchmark is a measurement of the performance of a very small piece of code, something that might take microseconds (µs) or nanoseconds (ns) to run
- It provides very precise timings, making it possible to compare operations that only take a tiny amount of time. For example, the following code compares the speed of three ways of computing a square root.

speed vs. accuracy

```
> library(microbenchmark)
> x = runif(1000)
> microbenchmark(sqrt(x),x^0.5,exp(0.5*log(x)))
Unit: microseconds
               expr    min      lq     mean  median      uq    max neval cld
            sqrt(x)  4.379  4.5280  7.62580  7.3755  7.5765 71.789   100 a
              x^0.5 26.990 27.2045 29.20157 30.1425 30.3490 42.836   100   c
 exp(0.5 * log(x)) 15.196 15.4210 17.61316 18.4675 18.7420 29.355   100  b
```

1M microsecond = 1 second

Example: for loop versus build-in function

```
cumsum_1 = function(x){
  for(i in 2:length(x)){
    x[i] = x[i] + x[i-1]
  }             slo
  return(x)      w
}
```

sapply will simplify list to vector or array

```
cumsum_2 = function(x){
    sapply(2:length(x), function(i) x[i] <<- x[i] + x[i-1] )
    return(x)
}
```

<<- is used to modify the input value as the output
it can look for variable outside of the function

```
> x = rep(1,length=1000)
> microbenchmark(my_cumsum_1(x), my_cumsum_2(x),cumsum(x))
Unit: microseconds
         expr      min         lq      mean    median       uq      max neval cld
  cumsum_1(x)  809.002  890.733 1024.36413  952.0435 1105.4545 2611.256   100   b
  cumsum_2(x) 1160.065 1289.626 1542.70321 1421.2905 1554.9715 3211.737   100    c
    cumsum(x)    2.269    3.182    3.83002    3.6445    4.1175    8.915   100  a
```

Example: sapply versus apply

```
> x = matrix(rnorm(1e6),nrow=100,ncol=1e4)
>
> microbenchmark(
+   y1 <- sapply(1:1e4,function(i) cumsum(x[,i])),
+   y2 <- apply(x,2,cumsum)
+ )

Unit: milliseconds
                                                 expr      min       lq     mean   median
 y1 <- sapply(1:10000, function(i) cumsum(x[, i])) 37.81795 44.84013 71.25728 49.55951
                           y2 <- apply(x, 2, cumsum) 40.64251 53.01257 85.55411 61.81404
       uq      max neval cld
 121.7896 150.2417   100  a
 132.0372 163.9973   100   b
> all(y1==y2)
[1] TRUE
```

Better implementation ?

```
library(Rcpp)

cppFunction('void cumsum_3(NumericVector& x){
          for(int i = 0; i<x.size(); i++){
              x[i] = x[i] + x[i-1];
          }
}')

> microbenchmark(cumsum_1(rep(1,length=1000)),
+                cumsum_2(rep(1,length=1000)),
+                cumsum_3(rep(1,length=1000)),
+                cumsum(rep(1,length=1000)),times=1000)
Unit: microseconds
                          expr      min       lq        mean    median
 cumsum_1(rep(1, length = 1000))  766.112  875.300 1076.159666  929.0250
 cumsum_2(rep(1, length = 1000)) 1118.403 1250.622 1439.504496 1328.0435
 cumsum_3(rep(1, length = 1000))    5.854    6.868    8.420184    7.7005
   cumsum(rep(1, length = 1000))    3.677    4.739    5.399996    5.0330
       uq        max neval cld
  998.760  76564.592  1000   b
 1424.347   3420.567  1000    c
    9.371     26.128  1000 a
    5.568     38.861  1000 a
```

>x=c(1,2,3)
>cumsum(x) pass by value
x
1,2,3
>cumsum_3(x) in Rcpp always pass by reference
x
1, 3,6

High Performance Functions With Rcpp

This magic comes by way of the Rcpp package, a fantastic tool makes it very simple to connect `C++` to `R`
Typical bottlenecks that `C++` can address include:

- Loops that can not be easily vectorised because subsequent iterations depend on previous ones.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in `C++` is much lower than that in `R`
- Problems that require advanced data structures and algorithms that `R` does not provide. Through the standard template library (STL), `C++` has efficient implementations of many important data structures, from ordered maps to double-ended queues.

Prerequistes

All examples in this lecture need version 0.10.1 or above of the `Rcpp` package. This version includes `cppFunction()` and `sourceCpp()`, which makes it very easy to connect `C++` to `R`. Install the latest version of `Rcpp` from CRAN with `install.packages("Rcpp").` You?ll also need a working `C++` compiler. To get it:

- On Windows, install Rtools `https://cran.r-project.org/bin/windows/Rtools/`.
- On Mac, install Xcode from the app store.
- On Linux, sudo apt-get install r-base-dev or similar.

cppFunction()

It allows you to write C++ functions in R

```
library(Rcpp)

cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
# add works like a regular R function
add
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x7f2f4aa933d0>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function.
We will summarize the basics by translating simple R functions to their C++ equivalents. We start simple with a function that has no inputs and a scalar output, and then get progressively more complicated:

No inputs, scalar output

The R function is

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
  return 1;
}
```

We can compile and use this from R with `cppFunction()`

```
cppFunction('int one() {
  return 1;
}')
```

Difference between C++ and R functions

This small function illustrates a number of important differences between `R` and `C++`:

- The syntax to create a function looks like the syntax to call a function; you do not use assignment to create functions as you do in `R`.
- You must declare the type of output the function returns. This function returns an int (a scalar integer). The classes for the most common types of `R` vectors are: `NumericVector`, `IntegerVector`, `CharacterVector`, and `LogicalVector`.
- Scalars and vectors are different. The scalar equivalents of `numeric`, `integer`, `character`, and `logical` vectors are: `double`, `int`, `String`, and `bool`.
- You must use an explicit return statement to return a value from a function.
- Every statement is terminated by a `;`.

Scalar input, scalar output

A scalar version of the `sign()` function which returns 1 if the input is positive, and -1 if it's negative:

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction('int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
```

Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower in C++. For example, we could implement the sum function in R using a loop.

In C++, loops have very little overhead. In STL, you will see alternatives to for loops that more clearly express your intent; they are not necessarily faster, but they can make your code easier to understand.

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

vector: always pass the reference of x, will not pass by value;
will not create the identical copy of x , but pass the address, so the value of x can be modified.

```
cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
```

Peformance

```
> x <- runif(1e6)
> microbenchmark(
+   sum(x),
+   sumC(x),
+   sumR(x)
+ )
Unit: microseconds
    expr        min          lq        mean      median          uq
  sum(x)    801.508    811.3270    892.7033    857.7995    908.4275
 sumC(x)    798.635    810.9305    870.4631    849.3895    892.4640
 sumR(x) 287038.151 295643.4335 302012.5137 298393.2220 301656.9130
        max neval cld
   1897.680   100   a
   1256.913   100   a
 369253.282   100    b
```

Vector input, vector output

Create a function that computes the Euclidean distance between a value and a vector of values

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

Not obvious that whether `x` is a scalar or not from the function definition. We need to make that clear in the documentation. That is not a problem in the `C++` version because we have to be explicit about types:

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}')
```

Performance

```
> y = runif(1e6)
> microbenchmark(pdistR(0,y),pdistC(0,y),times=10000L)
Unit: milliseconds
         expr      min       lq     mean   median       uq     max
 pdistR(0, y) 5.442920 6.218503 7.393230 6.511349 7.735468 99.9601
 pdistC(0, y) 4.101499 4.982845 6.002427 5.177412 6.155853 95.7577
 neval cld
 10000   b
 10000  a
```

Note that because the `R` version is fully vectorised, it is already going to be fast. Assuming it took you 10 minutes to write the `C++` function, you need to run it $600,000$ times to make rewriting worthwhile.

The reason why the `C++` function is faster is subtle, and relates to memory management. The `R` version needs to create an intermediate vector the same length as `y (x - ys)`, and allocating memory is an expensive operation. The `C++` function avoids this overhead because it uses an intermediate scalar.

Matrix input, vector output

Each vector type has a matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix`, and `LogicalMatrix`. Using them is straightforward.

```
> cppFunction('NumericVector rowSumsC(NumericMatrix x) {
+   int nrow = x.nrow(), ncol = x.ncol();
+              NumericVector out(nrow);
+              for (int i = 0; i < nrow; i++) {
+              double total = 0;
+              for (int j = 0; j < ncol; j++) {
+              total += x(i, j);
+              }
+              out[i] = total;
+              }
+              return out;
+              }')
> set.seed(2015)
> x = matrix(sample(100), 10)
> microbenchmark(rowSums(x),rowSumsC(x),times=10000L)
Unit: microseconds
        expr   min    lq     mean median    uq      max neval cld
  rowSums(x) 3.189 3.810 5.025323  4.109 4.458 6060.706 10000   b
 rowSumsC(x) 1.543 1.833 2.410042  1.978 2.156  661.357 10000  a
```

The main differences: (1) In `C++`, you subset a matrix with `()`, not `[]`. (2) Use `.nrow()` and `.ncol()` methods to get the dimensions of a matrix.

Using sourceCpp()

Inline `C++` with `cppFunction()`. This makes presentation simpler, but for real problems, it is usually easier to use stand-alone `C++` files and then source them into `R`using `sourceCpp()`.
This lets you take advantage of text editor support for `C++` files (e.g., syntax highlighting) as well as making it easier to identify the line numbers in compilation errors.
Your stand-alone `C++` file should have extension `.cpp`, and needs to start with:

```
#include <Rcpp.h>
using namespace Rcpp;
```

Using sourceCpp()

To compile the C++ code, use `sourceCpp("path/to/file.cpp")`. This will create the matching Rfunctions and add them to your current session.

Note that these functions can not be saved in a `.Rdata` file and reloaded in a later session; they must be recreated each time you restart R.

```
#include <Rcpp.h>
using namespace Rcpp;
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}
```

```
> sourceCpp("~/Dropbox/Umich/Biostat_615_Material/Fall_2015/Rcode/example_mean.cpp")
> x = runif(1e6)
> microbenchmark(meanC(x),mean(x))
Unit: microseconds
     expr      min       lq      mean    median       uq     max neval cld
 meanC(x)  798.977  801.792  866.0348  804.8845  869.633 1178.39   100   a
  mean(x) 1601.225 1605.066 1755.3828 1638.4170 1710.309 3455.01   100    b
```

Attributes and other classes

All Robjects have attributes, which can be queried and modified with `R.attr()`. `Rcpp` also provides `.names()` as an alias for the name attribute.

```
> cppFunction('NumericVector attribs() {
+   NumericVector out = NumericVector::create(1, 2, 3);
+
+             out.names() = CharacterVector::create("a", "b", "c");
+             out.attr("my-attr") = "my-value";
+             out.attr("class") = "my-class";
+
+             return out;
+             }')
> attribs()
a b c
1 2 3
attr(,"my-attr")
[1] "my-value"
attr(,"class")
[1] "my-class"
```

Lists and data frames

- `Rcpp` also provides classes `List` and `DataFrame`, but they are more useful for output than input.
- This is because lists and data frames can contain arbitrary classes but `C++` needs to know their classes in advance.
- If the list has known structure, you can extract the components and manually convert them to their `C++` equivalents with `as()`.
- For example, the object created by `lm()`, the function that fits a linear model, is a list whose components are always of the same type.

Lists and data frames: Example

- The following code illustrates how you might extract the mean percentage error (`mpe()`) of a linear model.

```
cppFunction('double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

            NumericVector resid = as<NumericVector>(mod["residuals"]);
            NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

            int n = resid.size();
            double err = 0;
            for(int i = 0; i < n; ++i) {
            err += resid[i] / (fitted[i] + resid[i]);
            }
            return err / n;
            }')
x = rnorm(1000)
y = 1+2*x+rnorm(1000)
fit = lm(y~x)
mpe(fit)
```

- Note the use of `.inherits()` and the `stop()` to check that the object really is a linear model.

You can put R functions in an object of type Function. This makes calling an R function from C++ straightforward.

```
> cppFunction(
+ 'RObject callfun(Function f, int n) {
+                 return f(n);
+             }'
+ )
> microbenchmark(callfun(seq_along,100),seq_along(100))
Unit: nanoseconds
                   expr   min      lq     mean
 callfun(seq_along, 100) 13964 15229.5 18679.84
         seq_along(100)    72    93.0   122.53
 median      uq   max neval cld
  16029 17014.5 79281   100   b
    107   149.0   387   100  a
```

What type of object does an R function return? Use the catchall type `RObject`.