Portfolio 3

Sam Bowyer

Vectorisation

`R` is an interpreted language, so loops (particularly nested loops) can be very slow. Instead, it is often useful to use built-in vector operations, which tend to be written in C/C++ and hence are compiled and run much faster.

As example consider the two functions below that both compute the minimum value that the sign function takes from elements in a vector:

```
minsin1 <- function(x){
  m = Inf
  for (i in 1:length(x)){
    if (sin(x[i]) < m){}
      m = sin(x[i])
  }
  return (m)
}

minsin2 <-function(x) min(sin(x))
```

Then running these two functions on the same input, we find that the second, vectorised version runs much faster:

```
x = 1:1e7
system.time(minsin1(x))
```

```
##    user  system elapsed
##   0.685   0.000   0.686
```

```
system.time(minsin2(x))
```

```
##    user  system elapsed
##   0.118   0.028   0.147
```

Vectorisation is then particularly useful when we're working with matrices, or even higher dimensional arrays—more dimensions typically mean more nested loops in unvectorised code, greatly slowing things down. Consider the two functions below which sum the rows of a matrix before applying the previous `minsin` functions to the resulting vector.

```
minsin_matrix1 <- function(X){
  m = Inf
  for (i in 1:nrow(X)){
    total = 0
    for (j in 1:ncol(X)){
      total = total + X[i,j]
    }
    if (sin(total) < m){
      m = sin(total)
```

```r
    }
  }
  return (m)
}

minsin_matrix2 <- function(X) min(sin(rowSums(X)))
```

Again, by running these functions on the same data we find that vectorisation has drastically sped up execution.

```r
X = matrix(1:1e8, 1e3, 1e5)
system.time(minsin_matrix1(X))
```

```
##    user  system elapsed
##   2.507   0.004   2.514
```

```r
system.time(minsin_matrix2(X))
```

```
##    user  system elapsed
##    0.21    0.00    0.21
```

Useful Functions

`R` includes a variety of functions which help us perform operations on vectors.

`apply`

`apply(X, MARGIN, FUN, ...)`: applies the function `FUN` to an array of dimension 2 or more using the dimensions given in the list `MARGIN` (in which 1 represents rows, 2 represents columns, `c(1,2)` represents both, etc.).

```r
x <- matrix(1:12, 3, 4)
print(x)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
apply(x, c(1,2), minsin2)
```

```
##           [,1]       [,2]      [,3]       [,4]
## [1,] 0.8414710 -0.7568025 0.6569866 -0.5440211
## [2,] 0.9092974 -0.9589243 0.9893582 -0.9999902
## [3,] 0.1411200 -0.2794155 0.4121185 -0.5365729
```

```r
apply(x, 1, minsin2)
```

```
## [1] -0.7568025 -0.9999902 -0.5365729
```

```r
apply(x, 2, minsin2)
```

```
## [1]  0.1411200 -0.9589243  0.4121185 -0.9999902
```

For an example with a 3-dimensional array:

```r
x <- array(1:12, c(2, 3, 2))
print(x)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

```
apply(x, 3, sum)
```

```
## [1] 21 57
```

```
apply(x, c(1,2), sum)
```

```
##      [,1] [,2] [,3]
## [1,]    8   12   16
## [2,]   10   14   18
```

```
apply(x, c(2,3), sum)
```

```
##      [,1] [,2]
## [1,]    3   15
## [2,]    7   19
## [3,]   11   23
```

```
apply(x, c(1,3), sum)
```

```
##      [,1] [,2]
## [1,]    9   27
## [2,]   12   30
```

lapply

`lapply(X, FUN, ...)`: works like `apply` but can be used on vectors and lists, and also returns a list.

```
lapply(1:4, sqrt)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
```

```
x <- list(a=1:3, b=c(TRUE, TRUE, FALSE), c=2:-1)
lapply(x, minsin2)
```

```
## $a
## [1] 0.14112
##
```

```
## $b
## [1] 0
##
## $c
## [1] -0.841471
```

Note that in the following code execution we find that `lapply` is slower than both vectorised code and a `for` loop.

```
func <- function(x) sqrt(x^2)
func_lapply <- function(x) lapply(x, func)
func_loop <- function(x){
  out = rep(NA, length(x))
  for (i in seq_len(length(x))){
    out[i] = func(x[i])
  }
  return(out)
}
x = 1:1e7
```

```
system.time(func(x))
```

```
##    user  system elapsed
##   0.028   0.020   0.048
```

```
system.time(func_lapply(x))
```

```
##    user  system elapsed
##   5.429   0.164   5.592
```

```
system.time(func_loop(x))
```

```
##    user  system elapsed
##   2.416   0.000   2.416
```

sapply

`sapply(X, FUN, ...)`: works like `sapply` but simplifies the output before returning.

```
sapply(1:4, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
```

```
x <- list(a=1:3, b=c(TRUE, TRUE, FALSE), c=2:-1)
sapply(x, minsin2)
```

```
##         a         b         c
##  0.141120  0.000000 -0.841471
```

mapply

`mapply(FUN, ...)`: the (potentially multiple) arguments given as `...` are used to run the function `FUN`.

```
mapply(sqrt, 1:4)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
```

```
mapply(function(x,y,z) x * y + z, c(1, 10, 100), c(2,3,4), c(0, 1, 2))
```

```
## [1]   2  31 402
```

Map

This works very similarly to `mapply`.

```
Map(rep, 1:3, 4:6)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2 2 2
##
## [[3]]
## [1] 3 3 3 3 3 3
```

Though it is ever so slightly faster.

```
system.time(Map(func, 1:1e7))
```

```
##    user  system elapsed
##   5.204   0.000   5.203
```

```
system.time(mapply(func, 1:1e7))
```

```
##    user  system elapsed
##   5.703   0.092   5.795
```

Reduce

`Reduce(FUN, X)` applies `FUN` to consecutive pairs of elements in a vector iteratively until a single element is left.

```
Reduce(rep, 1:3)
```

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

Above, `Reduce` has first executed `rep(1,2)` to obtain the vector `c(1,1)` and has then executed `rep(c(1,1), 3)` to obtain the output.

Below, `Reduce` is used to write the elements of a list as the digits in a number.

```
Reduce(function(a,b) 10*a + b, 1:6)
```

```
## [1] 123456
```

Filter

`Filter(FUN, X)` removes any elements from the vector `X` who do not evaluate to `TRUE` under the function `FUN`.

```
Filter(function(x) sqrt(x) %% 1 == 0, 1:30)
```

```
## [1]  1  4  9 16 25
```

```
Filter(function(x) sqrt(x^2) == x, -10:10)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Parallel Programming

For large tasks we can distribute computation across multiple CPU cores in the hopes of a speed increase (though this won't help if the task is slow for a non-CPU-related bottleneck,m such as within the task's use of memory, networking or I/O).

We can check how many cores our CPU has by using the `parallel` package:

```
library(parallel)
num_cores <- detectCores()
num_cores
```

```
## [1] 16
```

Consider the function below which squares the square-root of its input:

```
id <- function(X) sqrt(X)**2
X = 1:1e7
system.time(lapply(X, id))
```

```
##    user  system elapsed
##   4.907   0.056   4.970
```

We can use the `mclapply()` function to run this function on multiple cores and see the difference in execution time varying the number of cores.

```
# 2 Cores
system.time(mclapply(X, id, mc.cores=2))
```

```
##    user  system elapsed
##   4.303   0.812   2.879
```

```
# 4 Cores
system.time(mclapply(X, id, mc.cores=4))
```

```
##    user  system elapsed
##   4.956   1.098   2.283
```

```
# 8 Cores
system.time(mclapply(X, id, mc.cores=8))
```

```
##    user  system elapsed
##   5.921   2.103   2.036
```

```
# 16 Cores
system.time(mclapply(X, id, mc.cores=16))
```

```
##    user  system elapsed
##   7.199   4.403   1.776
```

Note that it is not necessarily true that using n cores will result in an n-factor speed-up, in fact in general the gains tend to diminish as n grows larger (for some tasks using too many cores may actually slow you down due to the time required for the core-allocation of computations to take place).

`forEach` and `doParallel`

Another way we can use parallelisation is with the `doParallel` package, which can be used with the (non-parallelised) package `foreach` which allows us to write loops of the following form:

```
library(foreach)
foreach(i=1:3) %do% {
  print(i)
}
```

```
## [1] 1
## [1] 2
```

```
## [1] 3
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

The `%do%` expression executes each loop sequentially but the `doParallel` package's `%dopar%` executes the loops in parallel. We must first load this package and register the desired number of cores (we'll use 2):

```
library(doParallel)
```

```
## Loading required package: iterators
```

```
registerDoParallel(2)
X = 1:1e6
```

And then we can compare the performance of `%do%` and `%dopar%` as follows:

```
system.time(foreach (i=1:10) %do% {lapply(X, id)})
```

```
##    user  system elapsed
##   4.463   0.072   4.536
```

```
system.time(foreach (i=1:10) %dopar% {lapply(X, id)})
```

```
##    user  system elapsed
##   0.412   0.117   2.714
```

(And finally we clean up the cluster.)

```
stopImplicitCluster()
```