Vectorisation in R

L. Gatto

November 1, 2014

Many operations in R are vectorized, and understanding and using vectorization is an essential component of becoming a proficient programmer.

R Gentleman in *R Programming for Bioinformatics*

Vectorisation

A **vectorised computation** is one that, when applied to a vector (of length greater than 1), automatically operates directly on all elements of the input vector.

```
(x <- 1:5)

## [1] 1 2 3 4 5

(y <- 5:1)

## [1] 5 4 3 2 1

x + y

## [1] 6 6 6 6 6
```

Recycling rule

What is `x` and `y` are of different length: the shorter vector is replicate so that its length matches the longer ones.

```
(x <- 1:6)

## [1] 1 2 3 4 5 6

(y <- 1:2)

## [1] 1 2

x+y

## [1] 2 4 4 6 6 8
```

If the shorter vector is not an even multiple of the longer, a warning is issued.

diff example (1)

Compute difference between times of events, `e`. Given `n` events, there will be `n-1` inter-event times. `interval[i] <- e[i+1] - e[i]`

Procedural implementation:

```
diff1 <- function(e) {
  n <- length(e)
  interval <- rep(0, n - 1)
  for (i in 1:(n - 1))
      interval[i] <- e[i + 1] - e[i]
  interval
}
e <- c(2, 5, 10.2, 12, 19)
diff1(e)

## [1] 3.0 5.2 1.8 7.0
```

diff example (2)

Vectorised implementation

```
diff2 <- function(e) {
  n <- length(e)
  e[-1] - e[-n]
}
e <- c(2, 5, 10.2, 12, 19)
diff2(e)

## [1] 3.0 5.2 1.8 7.0
```

```
all.equal(diff1(e), diff2(e))

## [1] TRUE
```

When using for loops

Initialising the result variable before iteration to avoid unnecessary copies at each iteration substantially increases performance.

```
f1 <- function(n = 5e3) {
    a <- NULL
    for (i in 1:n)
        a <- c(a, sqrt(i))
    a
}
system.time(f1())

##    user  system elapsed
##   0.047   0.001   0.049
```

```
f2 <- function(n = 5e3) {
    a <- numeric(n)
    for (i in 1:n)
        a[i] <- sqrt(i)
    a
}
system.time(f2())

##    user  system elapsed
##   0.005   0.000   0.004
```

*apply functions

How to apply a function, iteratively, on a set of elements?

`apply(X, MARGIN, FUN, ...)`

- `MARGIN` = 1 for row, 2 for cols.
- `FUN` = function to apply
- `...` = extra args to function.
- `simplify` = should the result be simplified if possible.

*apply functions are (generally) NOT faster than loops, but more succint and thus clearer.

Usage (1)

```
v <- rnorm(1000) ## or a list
res <- numeric(length(v))

for (i in 1:length(v))
  res[i] <- f(v[i])

res <- sapply(v, f)

## if f is vectorised
f(v)
```

Usage (2)

```
## M is a matrix/data.frame/array
rowResults <- numeric(nrow(M))
colResults <- numeric(ncol(M))

for (i in 1:nrow(M))
  rowResults <- f(M[i, ])

for (j in 1:ncol(M))
  colResults <- f(M[, j])

rowResults <- apply(M, 1, f)
colResults <- apply(M, 2, f)

rowSums(M)
colSums(M)
```

*apply functions

apply	matrices, arrays, data.frames
lapply	lists, vectors
sapply	lists, vectors
vapply	with a pre-specified type of return value
tapply	atomic objects, typically vectors
by	similar to tapply
eapply	environments
mapply	multiple values
rapply	recursive version of lapply
esApply	`ExpressionSet`, defined in `Biobase`

See also the `BiocGenerics` package for `[l|m|s|t]apply` S4 generics, as well as parallel versions in the `parallel` package.

See also the `plyr` package, that offers its own flavour of apply functions.

Anonymous functions

A function defined/called without being assigned to an identifier and generally passed as argument to other functions (and in particular `apply` functions).

```
M <- matrix(rnorm(100), 10)
apply(M, 1, function(Mrow) 'do something with Mrow')
apply(M, 2, function(Mcol) 'do something with Mcol')
```

Example - replicate

```
f <- function(d) {
  M <- matrix(runif(d^2), nrow=d)
  solve(M)
}
system.time(f(100))

##    user  system elapsed
##   0.002   0.000   0.002

res <- replicate(10, system.time(f(100))[["elapsed"]])
summary(res)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0010  0.0010  0.0010  0.0012  0.0010  0.0020
```

Example - tapply

```
dfr <- data.frame(A = sample(letters[1:5], 100,
                             replace = TRUE),
                  B = rnorm(100))
tapply(dfr$B, dfr$A, mean)
```

```
##           a           b           c           d
## -0.30394060 -0.44737268 -0.07489295  0.42696977  0.23125
```

```
tapply(dfr$B, dfr$A, summary)[1:2]
```

```
## $a
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.4330 -0.9577 -0.1193 -0.3039  0.1976  1.5010
##
## $b
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -3.3780 -0.8887 -0.4892 -0.4474  0.3622  1.8640
```

Efficient apply-like functions

- In `base`: rowSums, rowMeans, colSums, colMeans
- In `Biobase`: rowQ, rowMax, rowMin, rowMedias, ...
- In `genefilter`: rowttests, rowFtests, rowSds, rowVars, ...

Generalisable on other data structures, like `ExpressionSet` instances.

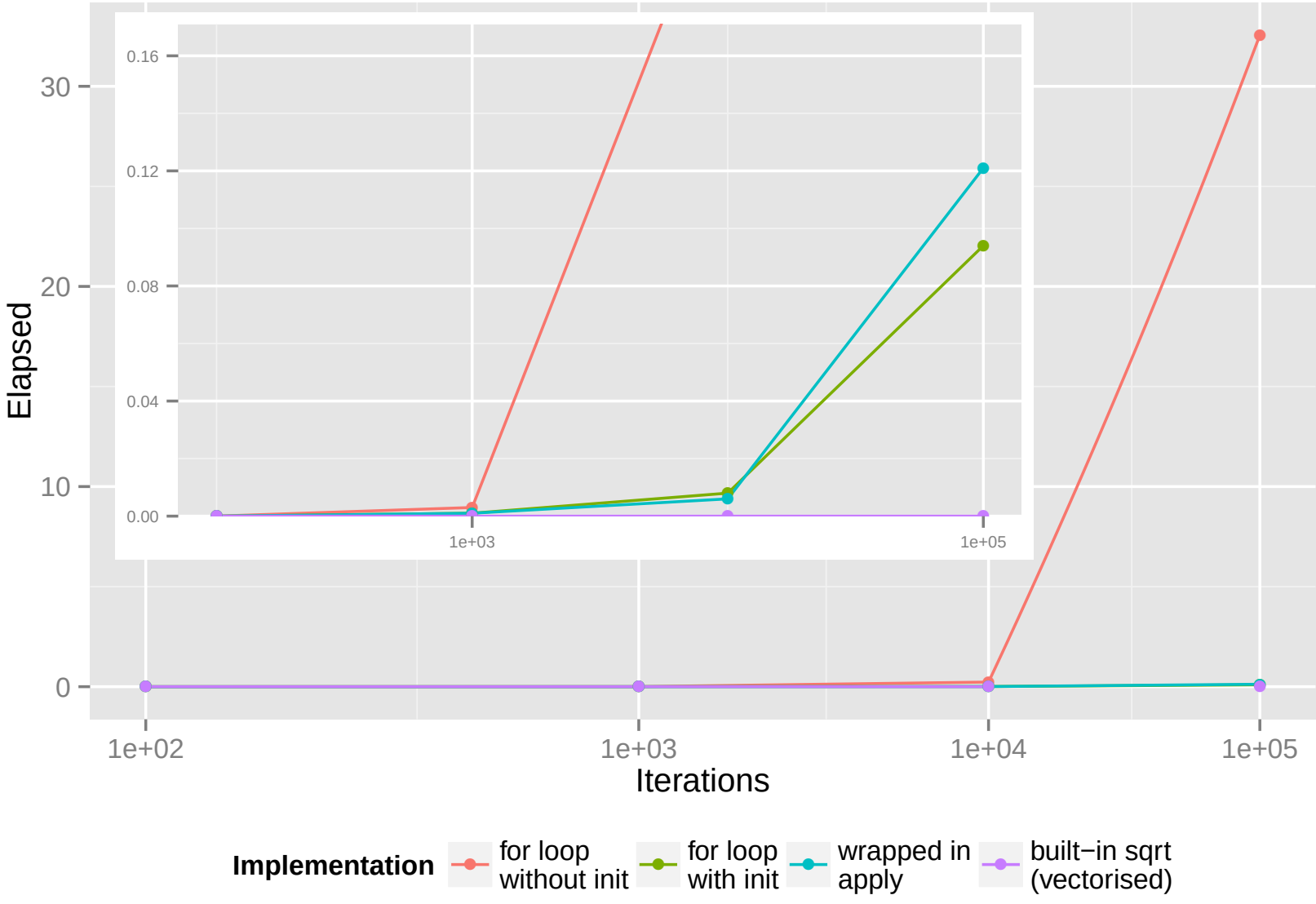
Timings (1)

```
f1 <- function(n) {
  a <- NULL
  for (i in 1:n) a <- c(a, sqrt(i))
  a
}
f2 <- function(n) {
  a <- numeric(n)
  for (i in 1:n) a[i] <- sqrt(i)
  a
}

f3 <- function(n)
  sapply(seq_len(n), sqrt)

f4 <- function(n) sqrt(n)
```

Timings (1)


Elapsed
30
20
10
0
0.16
0.12
0.08
0.04
0.00
1e+03
1e+05
1e+02
1e+03
1e+04
1e+05
Iterations
Implementation
for loop without init
for loop with init
wrapped in apply
built−in sqrt (vectorised)

Parallelisation

Vectorised operations are natural candidats for parallel execution.
See later, *Parallel computation* topic.