Benchmarking with R/SpaDES R 3.1.1

Eliot McIntire

October 7, 2014

The objective of this file is to show some speed comparisons between R and C++ and in some cases, other languages or software. Clearly this is NOT a comparison between R and C++ because many of the functions in R are written in C++ and wrapped in R. This document shows three important points:

1. built-in R functions (written in R or C++ or any other language) are often faster than ad hoc C++ functions because they are optimized.
2. built-in R functions are to be used in a vectorized way, avoiding loops unless it is strictly necessary to keep the sequence
3. there are often different ways to do the same thing in R; some are much faster than others

Tests are done on an HP Z400, Xeon 3.33 GHz processor, running Windows 7 Enterprise

Low level functionality

We will begin with low level functions that are generally highly optimized in R. As a result, the comparison C++ functions, which are not optimized, may not fully represent what C++ could do. However, this represents a real world issue: if the "out of the box" R function is competitive with a "quick" C++ version, then the R version is easier to write as there is no further development. If there is a desire or need for more speed, then a more optimzed C++ version can be written and used either in native C++ applications or R.

```r
library(data.table)
library(microbenchmark)
library(Rcpp)
library(numbers)
library(magrittr) # for pipe used below
```

```
##
## Attaching package: 'magrittr'
##
## The following object is masked from 'package:numbers':
##
##     mod
```

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:data.table':
##
##     between, last
##
## The following object is masked from 'package:stats':
```

```
##
##     filter
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
path <- "~/GitHub/SpaDES/SAMPLE/Functions in progress/"
setwd(path)
sourceCpp(file="meanCPP.cpp")
```

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

// [[Rcpp::export]]
double meanC1(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}

// [[Rcpp::export]]
double meanC2(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

// [[Rcpp::export]]
NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
}

// [[Rcpp::export]]
bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

// [[Rcpp::export]]
int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

// [[Rcpp::export]]
NumericVector pminC(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}

// [[Rcpp::export]]
int fibonacciC(const int x) {
    if (x == 0 || x == 1) return(x);
    return (fibonacciC(x - 1)) + fibonacciC(x - 2);
}

// [[Rcpp::export]]
int fibC(int n) {
    return n < 2 ? n : fibC(n-1) + fibC(n-2);
}
```

For the mean, I show two different C++ versiopns. The R function, "mean" is somewhat slower (1/2x), but the .Primitive option in R, sum/length is faster than either C++ function.

Mean

```
## Unit: microseconds
##                    expr    min     lq   mean median     uq   max
##          a <- meanC1(x)  86.94  90.93  94.98  92.77  97.23 143.5
##          d <- meanC2(x) 614.39 640.50 648.66 643.27 646.95 739.4
##            b <- mean(x) 182.47 191.08 201.90 196.91 210.28 296.1
##    e <- mean.default(x) 170.80 177.87 182.96 180.48 183.70 225.2
##   g <- sum(x)/length(x)  85.40  89.39  94.03  90.93  93.23 197.8
##  i <- .Internal(mean(x))  168.04  174.79  177.69  175.72  178.33   209.8
##        h <- rowMeans(x1) 1063.51 1204.82 2339.54 1271.33 1480.84 43654.1
##  neval cld
##    100  a
##    100  a
##    100  a
##    100  a
##    100  a
##    100  a
##    100   b
```

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

Minimum of pair of numbers Below, we take the minimum of each of a pair of columns.

```
x2 <- rnorm(1e5)
rm(a, b, d)
(mb<-microbenchmark(a<-pminC(x,x2),b<-base::pmin(x,x2), d<-.Internal(pmin(x,x2))))
```

```
## Unit: nanoseconds
##                         expr     min      lq    mean  median      uq
##            a <- pminC(x, x2) 1206358 1413101 4585149 2367712 2837261
##       b <- base::pmin(x, x2) 2845862 2888409 3676234 2928805 3023422
##  d <- .Internal(pmin(x, x2))       0     307    1588    1536    2458
##       max neval cld
##  51723476   100   b
##  57963237   100   b
##      5837   100  a
```

```
print(pmins<-round(summary(mb)[[4]][1]/min(summary(mb)[[4]][3]),0))
```

```
## [1] 2887
```

```
all.equal(a,b,d)
```

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

The internal R function is ***2887x*** faster than the C++ version, or the base R version.

This is taken from a blog post by Wingfeet at http://www.r-bloggers.com/quicksort-speed-just-in-time-compiling-and-vectorizin which drew on benchmark tests here: http://julialang.org/ Essentially, this was a benchmark to test the speed of Julia. It shows for the Quicksort, that R is 524x slower than C. Below is a "simple" version, then the best, fastest version that Wingfeet was able to do. But, there was no explicit comparison of how the base R sort would match with C.

Sorting Real number sorting:

```r
x = runif(1e5)
xtbl <- tbl_df(data.frame(x=x))
(mb <- microbenchmark(a0 <- qsort(x), a <- wfqsx(x), b <- wfqs1(x),
                      d <- sort(x),
                      e <- sort(x, method="quick"),
                      f <- .Internal(sort(x,decreasing = FALSE)),
                      g <- data.table(x=x,key="x"), h<-arrange(xtbl,x),
                      times=1L))
```

```
## Unit: milliseconds
##                                         expr      min       lq     mean
##                               a0 <- qsort(x) 3333.017 3333.017 3333.017
##                                a <- wfqsx(x) 1173.712 1173.712 1173.712
##                                b <- wfqs1(x) 1005.015 1005.015 1005.015
##                                 d <- sort(x)   13.153   13.153   13.153
##               e <- sort(x, method = "quick")    8.790    8.790    8.790
##  f <- .Internal(sort(x, decreasing = FALSE))   11.372   11.372   11.372
##            g <- data.table(x = x, key = "x")    9.344    9.344    9.344
##                         h <- arrange(xtbl, x)   60.860   60.860   60.860
##    median       uq      max neval
##  3333.017 3333.017 3333.017     1
##  1173.712 1173.712 1173.712     1
##  1005.015 1005.015 1005.015     1
##    13.153   13.153   13.153     1
##     8.790    8.790    8.790     1
##    11.372   11.372   11.372     1
##     9.344    9.344    9.344     1
##    60.860   60.860   60.860     1
```

```r
print(sumReals<-round(summary(mb)[[4]][1]/min(summary(mb)[[4]][4:7]),0))
```

```
## [1] 379
```

```r
all.equalV(a0,a,b,d,e,f,g$x,h$x)
```

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

And Integers are faster in the low-level R functions:

```r
x = sample(1e6,size = 1e5)
xtbl <- tbl_df(data.frame(x=x))
(mb <- microbenchmark(a0 <- qsort(x), a <- wfqsx(x), b <- wfqs1(x),
                      d <- sort(x),
                      e <- sort(x, method="quick"),
                      f <- .Internal(sort(x,decreasing = FALSE)),
                      g <- data.table(x=x,key="x"), h<-arrange(xtbl,x),
                      times=1L))
```

```
## Unit: milliseconds
##                                         expr      min       lq     mean
##                               a0 <- qsort(x) 3289.763 3289.763 3289.763
##                                a <- wfqsx(x) 1140.186 1140.186 1140.186
##                                b <- wfqs1(x)  930.146  930.146  930.146
##                                 d <- sort(x)   10.878   10.878   10.878
##               e <- sort(x, method = "quick")    6.780    6.780    6.780
##  f <- .Internal(sort(x, decreasing = FALSE))   10.425   10.425   10.425
##            g <- data.table(x = x, key = "x")    4.609    4.609    4.609
##                        h <- arrange(xtbl, x)   27.377   27.377   27.377
##    median       uq      max neval
##  3289.763 3289.763 3289.763     1
##  1140.186 1140.186 1140.186     1
##   930.146  930.146  930.146     1
##    10.878   10.878   10.878     1
##     6.780    6.780    6.780     1
##    10.425   10.425   10.425     1
##     4.609    4.609    4.609     1
##    27.377   27.377   27.377     1
```

```
print(sumInts <- round(summary(mb)[[4]][1]/min(summary(mb)[[4]][4:7]),0))
```

```
## [1] 714
```

```
all.equalV(a0,a,b,d,e,f,g$x,h$x)
```

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

The first three function are 3 different implementations of the quicksort algorithm shown on the Julia pages, with the first, qsort, being the one that the Julia testers used. Using the data.table sorting we were able to achieve ***483x*** speedup if Reals, and **714x** speedup if integers. These put them as fast or faster than C or Fortran or Julia. In Wingfeet's blog post, he also showed that using JIT can speed up non-optimized, "procedural" R code, though not as fast as the low level functions that exist in various R packages.

```
fibR1 = function(n) {
    fib <- numeric(n)
    fib[1:2] <- c(1, 2)
    for (k in 3:n) {
        fib[k] <- fib[k - 1] + fib[k - 2]
    }
    return(fib)
}
fibR2 = function(n) {
     if (n < 2) {
         return(n)
     } else {
         return(fibR2(n-1) + fibR2(n-2))
     }
}

N = 20L
(mbFib <- microbenchmark(times=10L, a<-numbers::fibonacci(N, sequence=TRUE)[N],
                         b<- fibC(N+1), d<-fibonacciC(N+1), e<-fibR1(N)[N], f<-fibR2(N+1)))
```

Fibonacci

```
## Unit: microseconds
##                                            expr      min       lq     mean
##  a <- numbers::fibonacci(N, sequence = TRUE)[N]    78.95    80.79   118.02
##                                b <- fibC(N + 1)    53.15    54.07    60.64
##                          d <- fibonacciC(N + 1)    57.75    58.37    60.49
##                                e <- fibR1(N)[N]    41.47    43.93    52.07
##                               f <- fibR2(N + 1) 86220.33 86550.57 87196.42
##    median       uq      max neval cld
##    108.90   142.54   223.02    10  a
##     61.13    66.66    68.20    10  a
##     59.75    60.21    71.58    10  a
##     47.00    66.05    69.12    10  a
##  86908.61 87247.29 90412.02    10   b
```

```
all.equalV(a,b,d, e, f)
```

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

Here, one of the two native R implementations is **1x faster** by pre-allocating the output vector size. The fibonacci function in the package numbers was 2x slower than the faster RR function because it has error checking. ***The native C++ version was 0.86x slower***.

Loops Loops have been the achilles heel of R in the past. In version 3.1 and forward, much of this appears to be gone. As could be seen in the fibonacci example, pre-allocating a vector and filling it up inside a loop can now be very fast and efficient in native R. To demonstrate these points, below are 6 ways to achieve the same result in R, beginning with a naive loop approach, and working up to the fully vectorized approach. I am using a very fast vectorized function, seq_len, to emphasize the differences between using loops and optimized vectorized functions.

```
N = 2e4

(mb = microbenchmark(times=4L,
naiveVector <- {
  set.seed(104)
  a <- numeric()
    for (i in 1:N) {
      a[i] = runif(1)+1
    }
   a
  } ,
presetVector1 <- {
    set.seed(104)
    norms <- runif(N)
    # pre-allocating vector length, generating normal random numbers once in each loop
    b <- numeric(N)
    for (i in 1:N) {
      b[i] = norms[i]+1
    }
    b
  },
```

```r
presetVector2 <- {
      set.seed(104)
      b <- runif(N)
      sapply(b,function(x) x)
      },
presetVector3 <- {
      set.seed(104)
      # pipe operator means that no intermediate objects are created
      num <- numeric(1)
      b <- runif(N) %>%
        sapply(.,function(x) x)
      },
vectorized1 <- {
  # vectorized with intermediate object
    set.seed(104)
    norms <- runif(N)
    d <- norms
    d
  },
vectorized2 <- {
  set.seed(104)
  # no intermediate object
  runif(N)
  }

))
```

```
## Unit: microseconds
##
##                            naiveVector <- {     set.seed(104)     a <- numeric()     for (i in 1:N) {
##   presetVector1 <- {     set.seed(104)     norms <- runif(N)     b <- numeric(N)     for (i in 1:N) {
##                                                           presetVector2 <- {     set.seed(104)     b <
##                                     presetVector3 <- {     set.seed(104)     num <- numeric(1)     b <
##                                                                 vectorized1 <- {     set.seed(104)
##                                                                                              vectori
##       min       lq     mean   median       uq      max neval cld
##  625688.0 633216.4 655461.9 645731.0 677707.4 704697.7     4   c
##   30896.9  31374.8  43538.7  31896.5  55702.7  79465.0     4  b
##   16311.8  17055.9  17772.8  18052.6  18489.7  18674.2     4 ab
##   17642.9  17748.3  18856.7  18423.5  19965.2  20937.0     4 ab
##     768.3    769.4    771.8    770.9    774.1    776.9     4 a
##     768.9    772.3    780.6    779.8    788.9    793.8     4 a
```

```r
all.equalV(naiveVector, presetVector1, presetVector2, presetVector3, vectorized1, vectorized2)
```

```
## Warning: coercing argument of type 'character' to logical

## [1] NA
```

```r
print(sumLoops <- round(summary(mb)[[4]][1]/summary(mb)[[4]][length(summary(mb)[[4]])],0))
```

```
## [1] 840
```

The fully vectorized function is ***840x*** faster than the fully naive loop. Note also that making as few intermediate objects as possible is faster as well. Comparing vectorized1 and vectorized2 shows an improvement of 106%. **Preallocating a vector** improved by ***15x***. Using pipes instead of inter

Conclusions In all cases shown here, the fastest native R function is faster than a simple (unoptimized) C or C++ function. As with any language, there is faster code and slower code. With R, it may take a few tries, but there is usually a very fast option.

Clearly, low level speed can be achieved within R, often better than quick implementations in C or C++. This is because efforts have been made in primitives and internal functions with core R functions to provide optimal versions, without any extra user coding. Many work flows to not require explicit loops. R's vectorization model allows for fast code, with little coding. ***Write vectorized code in R***

High level functionality

R also has numerous high level functions and packages that allow users to do a diversity of analyses and data manipulations, from GIS to MCMC to optimally stored file-based object storing for fast access (ff package), and much more. Here are a few examples.

```
  library(raster)
  lcc05 <- raster("c:/shared/data/LCC2005_V1_4a.tif")
  age <- raster("c:/shared/data/age.tif")
```

```
ext1 <- extent(-1073154,-987285,7438423,7512480) # small central Sask 100 Thousand
vegMapLcc1 <- crop(lcc05,ext1)

ext2 <- extent(1612240, 1895057, 6756615, 6907451) # small 600k pixels Quebec City Lac St. Jean
vegMapLcc2 <- crop(lcc05,ext2)

ext3 <- extent(-1380607, -345446, 7211410, 7971750) # large central BC 12Million
vegMapLcc3 <- crop(lcc05,ext3)
```

GIS

```
## Loading required package: snow
```

```
## 12 cores detected
```

```
## Using cluster with 12 nodes
## Using cluster with 12 nodes
## Using cluster with 12 nodes
```

```
## Unit: milliseconds
##                                                                                  expr
##  vegMapLcc1.crsAge <- projectRaster(vegMapLcc1, crs = crs(age),      method = "ngb")
##  vegMapLcc2.crsAge <- projectRaster(vegMapLcc2, crs = crs(age),      method = "ngb")
##  vegMapLcc3.crsAge <- projectRaster(vegMapLcc3, crs = crs(age),      method = "ngb")
##      min      lq    mean  median      uq     max neval
##    797.4   797.4   797.4   797.4   797.4   797.4     1
##   3855.2  3855.2  3855.2  3855.2  3855.2  3855.2     1
##  26826.5 26826.5 26826.5 26826.5 26826.5 26826.5     1
```

```
## Unit: milliseconds
##                                          expr   min    lq  mean median
##  age1.crsAge <- crop(age, vegMapLcc1.crsAge) 129.3 129.3 129.3  129.3
##  age2.crsAge <- crop(age, vegMapLcc2.crsAge) 136.5 136.5 136.5  136.5
##  age3.crsAge <- crop(age, vegMapLcc3.crsAge) 283.9 283.9 283.9  283.9
##     uq   max neval
##  129.3 129.3     1
##  136.5 136.5     1
##  283.9 283.9     1
```

```
## 12 cores detected
```

```
## Warning: closing unused connection 16 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 15 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 14 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 13 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 12 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 11 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 10 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 9 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 8 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 7 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 6 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
## Warning: closing unused connection 5 (<-W-VIC-A105200.nrn.nrcan.gc.ca:11710)
```

```
## Using cluster with 12 nodes
## Using cluster with 12 nodes
## Using cluster with 12 nodes
```

```
## Unit: milliseconds
##                                                                            expr
##  ageMapSmall <- projectRaster(age1.crsAge, to = vegMapLcc1, method = "ngb")
##    ageMapMed <- projectRaster(age2.crsAge, to = vegMapLcc2, method = "ngb")
##     ageMapLg <- projectRaster(age3.crsAge, to = vegMapLcc3, method = "ngb")
##      min      lq    mean  median      uq     max neval
##    695.7   695.7   695.7   695.7   695.7   695.7     1
##   2027.2  2027.2  2027.2  2027.2  2027.2  2027.2     1
##  27511.0 27511.0 27511.0 27511.0 27511.0 27511.0     1
```

Since the raster package can run in “parallel” mode for some of its functions, this reproject raster function reprojected **12 million pixels** in ***27 seconds*** on a 6 core, hyperthreaded machine (shows up as 12 cores), with a peak 600MB RAM use per core (7.2GB).

Raster to Polygon

The version of the function to polygonize a raster that exists in the raster package is slow. Thanks to John Baumgartner (http://johnbaumgartner.wordpress.com/2012/07/26/getting-rasters-into-shape-from-r/), we can call a GDAL function, if we have python, GDAL core, and GDAL-python bindings installed. This speedup is on the order of 50x-100x, depending on what the output format is. In addition to the instructions on John's web page, there are a few things that need to be done now. First, I used python3.3 because the GDAL python bindings for Windows only existed for 3.3. The current version of python is 3.4.2, so I used a slightly older version for this to work. And, there are 3 things to install, python, gdal core, and gdal python bindings (available on the gdal web page).

```
(mbRasterToPoly <- microbenchmark(times=1L,
ageMapPolypy <- gdal_polygonizeR(ageMapSmall, outshape=file.path("c:","Temp","ageSmall")),
ageMapPolypyMed <- gdal_polygonizeR(ageMapMed, outshape=file.path("c:","Temp","ageMed")),
ageMapPolypyLg <- gdal_polygonizeR(ageMapLg, outshape=file.path("c:","Temp","ageLg"))
))
```

```
## Unit: seconds
##                                                                                                 expr
##  ageMapPolypy <- gdal_polygonizeR(ageMapSmall, outshape = file.path("c:",      "Temp", "ageSmall"))
##   ageMapPolypyMed <- gdal_polygonizeR(ageMapMed, outshape = file.path("c:",      "Temp", "ageMed"))
##      ageMapPolypyLg <- gdal_polygonizeR(ageMapLg, outshape = file.path("c:",      "Temp", "ageLg"))
##      min       lq     mean   median       uq      max neval
##    1.948    1.948    1.948    1.948    1.948    1.948     1
##   10.691   10.691   10.691   10.691   10.691   10.691     1
##  159.026 159.026  159.026  159.026  159.026  159.026     1
```

```
(mbRasterToPoly <- microbenchmark(times=1L,
  ageMapPoly <- rasterToPolygons(ageMapSmall, dissolve=TRUE),
  ageMapPolyMed <- rasterToPolygons(ageMapMed, dissolve=TRUE)
))
```

```
## Loading required package: rgeos
## rgeos version: 0.3-8, (SVN revision 460)
##  GEOS runtime version: 3.4.2-CAPI-1.8.2 r3921
##  Polygon checking: TRUE
```

```
## Unit: seconds
##                                                            expr    min
##   ageMapPoly <- rasterToPolygons(ageMapSmall, dissolve = TRUE)  49.38
##  ageMapPolyMed <- rasterToPolygons(ageMapMed, dissolve = TRUE) 289.35
##      lq   mean median     uq    max neval
##   49.38  49.38  49.38  49.38  49.38     1
##  289.35 289.35 289.35 289.35 289.35     1
```

```
#endCluster()
```

Links to other benchmark testing with R

Comparing data.table, dplyr, and pandas in Python: https://github.com/Rdatatable/data.table/wiki/Benchmarks-%3A-Grouping

Comparing data.table and dplyr for split-combine-apply style analysis: http://www.brodieg.com/?p=7