Enhance Your R Performance and Flexibility with Rcpp

Jon Meek

meekjt (at) gmail.com
meekj (at) ieee.org

Example code on USB sticks and at:
`https://github.com/meekj/RVAR-March2019`

11-March-2019 / RVA R Users

Overview

- I am not an expert, and other disclaimers...
- Range of native R performance: from horrible to HPC
- "Standard example" is 23 million observations of 5 variables, 813 MB
- Basic benchmarking techniques
- Extending and speeding-up R using C++ with Rcpp
- Today we will only look at single-thread methods
- A real (simple) Rcpp application
- Providing access to a common C library
- Quick mention of other performance issues

Three Ways to Increment a Vector with Base R: 1

```
> ## Allocate a 23 million point vector
> vlength <- 23e6
> vec <- vector(mode = 'numeric', length = vlength)
> str(vec)
 num [1:23000000] 0 0 0 0 0 0 0 0 0 0 ...

> incVal1 <- 1
> ## Use a loop to increment every element
> t_start <- proc.time()
> for (i in 1:length(vec)) {
+     vec[i] <- vec[i] + incVal1
+ }
> proc.time() - t_start
   user  system elapsed
  1.532   0.046   1.578
> str(vec)
 num [1:23000000] 1 1 1 1 1 1 1 1 1 1 ...
```

Three Ways to Increment a Vector with Base R: 2 & 3

```
   user  system elapsed
  1.532   0.046   1.578   for loop from method 1

> ## Do the loop another way
> vec[1:length(vec)] <- vec[1:length(vec)] + incVal1
   user  system elapsed
  0.306   0.140   0.446
> str(vec)
 num [1:23000000] 2 2 2 2 2 2 2 2 2 2 ...

> ## Use vectorized R method to increment every element
> vec <- vec + incVal1
   user  system elapsed
  0.054   0.060   0.114         "The right way"
> str(vec)
 num [1:23000000] 3 3 3 3 3 3 3 3 3 3 ...
```

Can we do Better?

- Use Julia for speed? Dirk Eddelbuettel says use Rcpp
- Rcpp provides an easy way to incorporate C++ into R code
- '`for`' & '`while`' loops in R are slow
    - vectorize if possible
    - if not possible use Rcpp
- Other uses for Rcpp
    - Integrate C/C++ libraries into R for your special requirement
    - Perform low-level bit-wise calculations
    - Communicate with hardware (sensors, lab equipment, etc)
    - Specialized computing where high performance is required
- Try Base R and common packages like dplyr first
- Using R + C++ is similar to how I used FORTRAN + Assembly and Pascal + Assembly in the far past

Simple Rcpp Code - In-line

```
library(Rcpp)
cppFunction('NumericVector incrementVector(double Increment,
                                           NumericVector TheData) {
   int n = TheData.size();  // C++ way to get length of vector
   for (int i = 0; i < n; ++i) {
     TheData[i] += Increment;
   }
   return TheData;
}')
```

```
> ## Use our simple in-line C++ function to increment every element
> vec <- incrementVector(incVal1, vec)
   user  system elapsed
  0.047   0.012   0.058
> str(vec)
 num [1:23000000] 4 4 4 4 4 4 4 4 4 4 ...
```

Running this a few times suggests only a minor improvement using C++
However...

Do Proper Benchmarking with microbenchmark

- Default is to run code block 100 times (after 2 warm-ups?)
- Result: Classes 'microbenchmark' and 'data.frame'
- Print method provides statistical analysis
- Columns can be added without affecting the print method
- Multiple tests can be combined into a data frame
- $expr contains the tested expression
- Individual measurements are in $time
- So, we can make boxplots, etc.
- Also built-in violin plot

Do Proper Benchmarking - R 3.5.x

Run each example 100 times - Ignore slow methods

```
library(microbenchmark)

> ## Base R - Fast Method
> mb_res1 <- microbenchmark(vec <- vec + incVal1)
> str(vec)  ## Note that we got another 100 increments
 num [1:23000000] 104 104 104 104 104 104 104 104 104 104 ...
> ## Rcpp
> mb_res2 <- microbenchmark(vec <- incrementVector(incVal1, vec))
> str(vec)
 num [1:23000000] 204 204 204 204 204 204 204 204 204 204 ...

> ## Look at structure of the microbenchmark result
> str(mb_res1)
Classes microbenchmark and data.frame: 100 obs. of  2 variables:
 $ expr: Factor w/ 1 level "vec <- vec + incVal1": 1 1 1 1 1 1 1 1 1 1 ...
 $ time: num  61967994 61090692 59822301 57745646 57785982 ...

> mb_res <- rbind(mb_res1, mb_res2) # Combine the benchmark results
> mb_res
Unit: milliseconds                 expr      min       lq     mean   median       uq      max neval
                 vec <- vec + incVal1 57.17667 59.81939 72.34597 60.48852 91.65781 196.0854   100
 vec <- incrementVector(incVal1, vec) 17.87807 18.11191 18.49772 18.31564 18.73056  23.2293   100
```

C++ provides about a 70 % reduction in median run time

Depending on R instance! (R version, OS version, compiler)

Do Proper Benchmarking - Nov 2017 - R 3.4.2

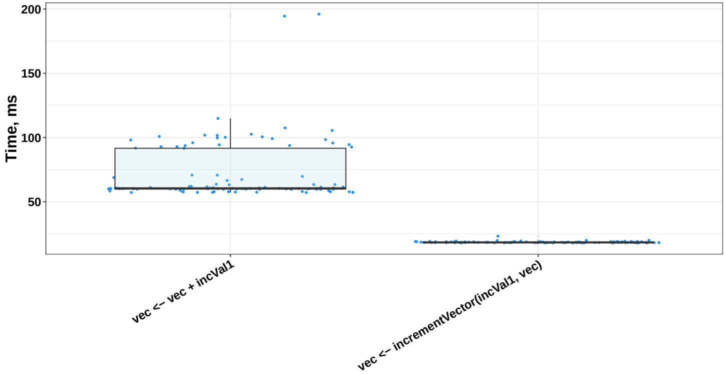
Run each example 100 times

```
library(microbenchmark)

vlength <- 23e6   # Allocate a 23 million point vector
vec <- vector(mode = 'numeric', length = vlength)

mb_res1 <- microbenchmark(
    for (i in 1:length(vec)) {
       vec[i] <- vec[i] + incVal1
    }
)

mb_res2 <- microbenchmark( vec[1:length(vec)] <- vec[1:length(vec)] + incVal1 )
mb_res3 <- microbenchmark( vec <- vec + incVal1 )
mb_res4 <- microbenchmark( vec <- incrementVector(incVal1, vec) )

rbind(mb_res1, mb_res2, mb_res3, mb_res4)
```

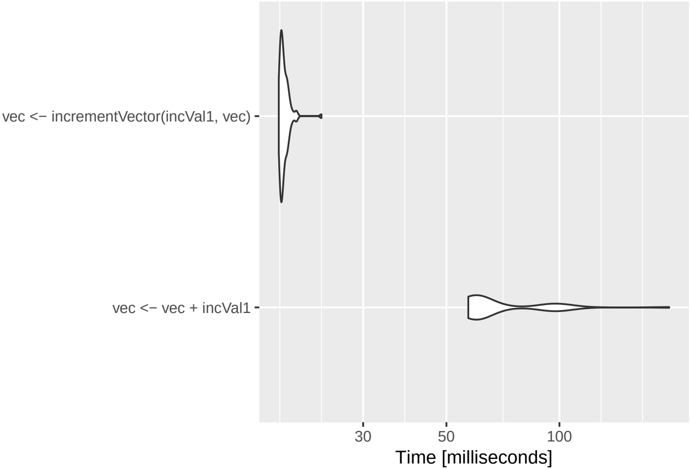
```
Unit: milliseconds
                                              expr        min         lq       mean     median         uq      max
for (i in 1:length(vec)) {vec[i] <- vec[i] + 1} 1187.43655 1189.21543 1193.26016 1190.90870 1193.85508 1232.245
   vec[1:length(vec)] <- vec[1:length(vec)] + 1  216.56800  217.42499  221.93871  218.30503  220.16905  338.374
                                   vec <- vec + 1   25.47441   26.33712   34.28377   26.80986   55.88667   57.498
                 vec <- incrementVector(1, vec)   17.27036   17.29239   17.77067   17.50366   18.22541   19.345

(neval = 100 column is cutoff)
```

C++ provides about a 35 % reduction in median run time

Incrementing Vector - Base and Rcpp / C++



microbenchmark Built-in plot



autoplot(mb_res)

A Real Rcpp Application - System or Network Utilization

- Questions often arise well after an “incident”
  - Why did something slow down, or break?
  - Too many users or sessions?
  - Too much bandwidth being consumed?
  - Was it due to YouTube traffic?
  - What time of day was the resource stressed? For how long?
- Per session log files typically retained for months
- Packet capture files are too large to retain for long
- Compute estimated throughput or concurrent sessions from network device log files
  - Millions, or a billion, records
  - Use session duration and end time
  - Distribute total bytes, active sessions, or unique users, across one second bins

Compute Estimated Throughput

~23 million log events covering 24 hours of "end times" (select columns)

```
Time Duration Status BytesSent BytesRecv
2015-04-13T23:57:49 49069 200 401 376
2015-04-13T23:57:49 256 200 522 132
2015-04-13T23:57:49 3063 200 527 3095
2015-04-13T23:57:49 376989 200 398 0
2015-04-13T23:57:49 540 200 766 132
2015-04-13T23:57:49 306792 200 402 0
2015-04-13T23:57:49 802 200 489 196339
...
```

- Use session duration to compute start time
- Distribute bytes received evenly across one second wide bins
- If duration $<=$ 1 s, full byte count goes in a single bin
- If duration $>$ 1 s, round up to spread across multiple bins
- Two nested loops: Each event; Fill appropriate bins
- R with `for` loops: 7 - 54 minutes (depending on R version!)
- Rcpp: (as low as) 670 milli-seconds !

Pre-process Data - Overview

- Reading raw ASCII data with readr is reasonably fast
- Preparing data with Base R & lubridate is very fast
- Simplified data; StartSecond is index / relative time

```
> head(events)
StartSecond Duration BytesRecv
1         2883    49.069       376
2         2932     0.256       132
3         2932     0.269       132
4         2932     0.253       132
5         2929     3.063      3095
6         2556   376.989         0
...
```

- Post-processing with Base R is very fast

Pure R Code - Quick Look

```
## Read and pre-process data...

## The Loop
for (i in 1:nrow(log_data)) {
    idx <- log_data$StartSecond[i] + 1                    # Start index; R starts at 1
    if (log_data$Duration[i] > 1) {                       # Does event span multiple bins?
        idt <- as.integer(ceiling(log_data$Duration[i])) # Event duration in bins
        bytes_per_second <- log_data$BytesRecv[i] / idt
        k <- idx + idt - 1                                # Final index to be incremented
        if ((k) > timerange_s) {                          # Don't go past end of vector
            idt <- timerange_s - idx
            k <- idx + idt
        }
        ccu[idx:k] <- ccu[idx:k] +  bytes_per_second     # Vectorized bin increments

    } else {
        ccu[idx] <- ccu[idx] + log_data$BytesRecv[i]     # Single bin to be incremented
    }
}

## Post processing
ccu <- 8 * ccu  / 1e3 # 8 bits / byte  - kbps

cca_df      <- data.frame(Throughput = ccu)              # Make it a dataframe
cca_df$Time <- MinTime + seconds(seq(1:nrow(cca_df)) - 1) # Add time column
```

Rcpp / C++ Code

C++ code in it's own file

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

// [[Rcpp::export]]
NumericVector concurrentEstimatedThroughput(int outlen, NumericVector StartSecond,
                                   NumericVector Duration, NumericVector Bytes) {

  NumericVector cca(outlen);        // Result will go in this vector
  int k, j, iduration, istart;
  int n = StartSecond.size();       // Number of events
  double bytes_per_second;

  for(int i = 0; i < n; ++i) {      // Process each event
    istart = int(StartSecond[i]);
    iduration = ceil(Duration[i]);  // Number of bins to increment

    if (iduration <= 1) {           // Just increment one bin
      cca[istart] += Bytes[i];
    } else {
      bytes_per_second = Bytes[i] / iduration; // Bytes per bin
      k = istart + iduration - 1;              // Last bin
      if (k >= outlen) {k = outlen - 1;}       // Don't go past end of vector
      for (j = istart; j <= k; j++) {          // Distribute bytes across bins
        cca[j] += bytes_per_second;            //  covering the event duration
      }
    }
  }
  return cca;
}
```

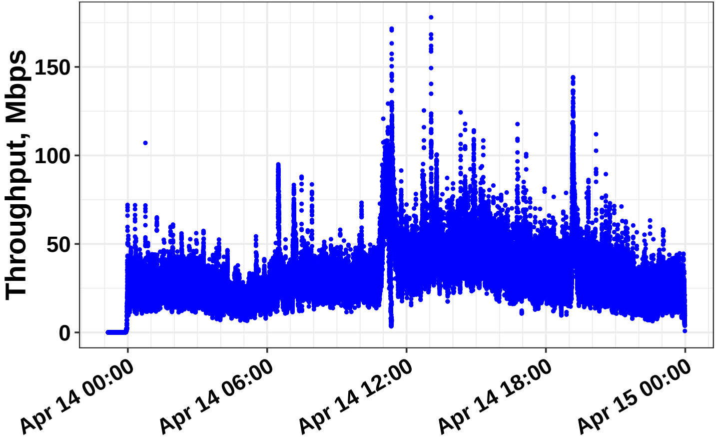
Compile and Run C++ Code

```
library(Rcpp)

myPath <- '~/wpl/talks/rvar-201903' # Adjust for local conditions

eventsDataFile <- paste0(myPath, '/events.rds')
codeFile       <- paste0(myPath, '/concurrent_activity.cpp')

events <- readRDS(eventsDataFile)
str(events)

## Compile  C++ code from a file
## rebuild & showOutput are optional and mostly useful when messing with compilers and optimization flags

sourceCpp(rebuild = TRUE, showOutput = TRUE, file=codeFile)

## Number of one second wide bins
##
timerange_s <- max(events$StartSecond) - min(events$StartSecond) + 1

## Run the C++ code with timing
##
t_end_prep <- proc.time()
cca <- concurrentEstimatedThroughput(timerange_s, events$StartSecond, events$Duration, events$BytesRecv)
t_end_loop <- proc.time()

t_end_loop - t_end_prep
```

Post-process & Plot Result

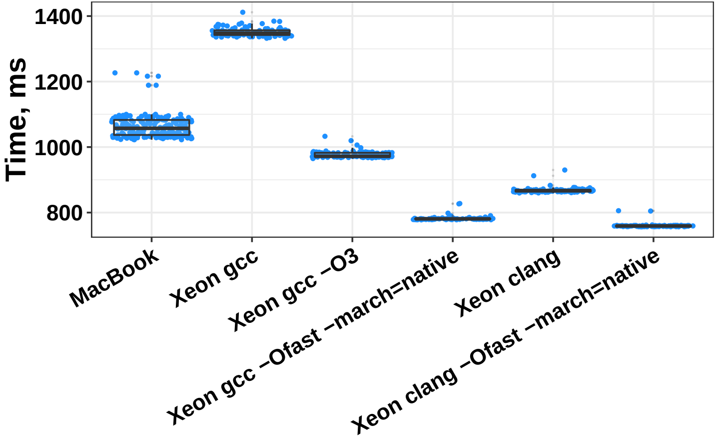
```
library(tidyverse)

str(cca) # Result vector

## Convert to data frame with relative time in seconds and throughput in Mbps
##
cca          <- 8 * cca  / 1e6                # 8 bits / byte  - Mbps
cca_df       <- data.frame(Throughput = cca)  # Make the vector a data frame
cca_df$Time <- seq(1:nrow(cca_df)) - 1       # Add relative time column

str(cca_df)

ggplot(cca_df) +
    geom_point(aes(x = Time, y = Throughput), size = 0.6, color = 'blue', shape = 19) +
    xlab('Time, seconds') + ylab('Throughput, Mbps')
```

The Result - Estimated Throughput



Estimated throughput for an Internet service over 24 hours with one second granularity.

MacBook vs “SuperWorkstation”, Estimated Throughput



It's mostly the compiler optimization flags, but gcc is slower for me in 2019.
(Fall 2017 results shown)

Selecting the Compiler & Flags for Rcpp

Warning: Can cause problems with package installation
Best to rename when not needed (mv Makevars off-Makevars)

```
cat ~/.R/Makevars
# CC=ccache clang-3.8 -Qunused-arguments
# CXX=ccache clang++-3.8 -Qunused-arguments
# CCACHE_CPP2=yes
# CC=clang-3.8 -Qunused-arguments
CXX=clang++
CXXFLAGS += -Ofast -march=native
# CXXFLAGS += -O3
```

Flags (not compiler) can be set in R:

```
Sys.setenv("PKG_CXXFLAGS"="-Ofast -march=native")
```

Packages have their own Makevars file

Compute Estimated Throughput - 23 million Events

	Read	Prep	Loop	Post
Base R (for loop)	12.0 s	0.6 s	6.8 - 54 minutes	0.06 s
Python / NumPy	128.6 s	0 s	15.6 minutes	0.33 s
Perl	98.3 s	9.6 s	6.6 minutes	0.30 s
R / C++	12.0 s	0.6 s	0.78 seconds	0.06 s

ASCII data read time includes date / time to seconds conversion
Shorter Base R loop time was for 3.5.2 on Intel NUC Core i7
Python result may not be fair, need to try Pandas
Python read & Prep are done in a single loop
readr is used to read data in R
Perl post time includes writing result
For larger data sets this is an "embarrassingly parallel" computation

Use Rcpp for Bit Level Computations

- IPAM (IPv4 Address Management)
- nbPtr <- nbReadAndLoadNetwork(network_description_file)
- nbLookupIPaddrs(nbPtr, vector_of_addresses)
- Finds “shortest” match
- `https://github.com/meekj/netblockr`

```
Example network description file:

10.16.0.0/12    NOAM xxx North America Supernet
10.16.0.0/22    NOAM PTN Princeton NJ Data Center Servers
10.16.8.0/23    NOAM PTN Princeton NJ West Wing Second Floor
10.18.12.0/23   NOAM SCV Sarah Creek VA
10.48.0.0/12    EMEA xxx EMEA Supernet
10.48.12.0/23   EMEA PSS Portsmouth Southsea
10.48.16.0/24   EMEA ZUR Zurich Wasserschopfi
```

Use Rcpp to Access C Library - libpcapR Package

- Set PKG_LIBS in environment or in Makevars file (`PKG_LIBS = -lpcap`)
- Add includes to C++ file as usual

```
#include <Rcpp.h>
using namespace Rcpp;
#include <pcap.h>
#include <stdio.h>
#include <string.h>
...
```

- Load network packet capture into a data frame using libpcap
  - Summarize traffic
  - Compute throughput with any time granularity
  - Currently focuses on header data rather than content
  - Supports IPv4 & IPv6
- `https://github.com/meekj/libpcapR`
- Requires libpcap-dev package to be installed.
- Package needs automated tests, vignette, etc and some users...
- Probably works only on Linux and Mac

Pre-made Rcpp Packages - Usually Performance Oriented

- dplyr and friends! (transparent use of Rcpp)
- AsioHeaders - Asynchronous network and low-level I/O
- BH - Boost peer-reviewed portable C++ source libraries via headers
- RcppArmadillo - Armadillo Templated Linear Algebra Library
- RcppGSL - GNU Scientific Library
- RcppBDT - Boost Date Time library
- Many, many others

Rcpp Resources

- I started here: Advanced R Programming by Hadley Wickham: `http://adv-r.had.co.nz/`
- Maybe a better starting point: `http://heather.cs.ucdavis.edu/Rcpp.pdf`
- Full book: Seamless R and C++ Integration with Rcpp by Dirk Eddelbuettel (Springer 2013)
- Rcpp Quick Reference: `https://cran.r-project.org/web/packages/Rcpp/vignettes/Rcpp-quickref.pdf`
- Rcpp Gallery: `http://gallery.rcpp.org/`
- Google $\rightarrow$ Stackoverflow are your friends, as expected

General R Performance Resources

- Efficient R Programming, Gillespie and Lovelace, (O'Reilly 2017)
- Performance chapter in Hadley's Advanced R Programming

C++ Notes

- C++ is a huge language
- Don't need to know a lot of C or C++ to benefit
- Be careful to not index past end of array, etc
- Lots of extensions/updates: C++11, C++14, C++17
- STL has really useful features (expandable containers, etc)
  - Use the std::vector<T> container and the .push_back(t) function to grow it
- Boost library
- Free C++ Annotations text (on-line & PDF): `http://www.icce.rug.nl/documents/cplusplus/`

Summary

- Use base R's vectorized functions when possible
- dplyr and other tidyverse packages are fast as well
- Avoid '`for`' & '`while`' when the loop count is high
- Use a recent version of R and packages
  - Performance can vary widely between R versions
  - Newer is not always faster, but sometimes it is much faster!
- Use Rcpp where appropriate
  - Compiler and flags can make a difference
  - 4000x performance improvements are possible
- Do benchmarking
- CPU clock speed may suggest how fast R executes base code
- Compiler and flags can have a significant impact on performance
- A busy desktop / laptop will have some effect

Other Performance Considerations

- Just In Time byte-code compiler enabled by default in R 3.4.0
- Use binary data formats (RDS, FST, Feather, netCDF, etc)
- Read large ASCII flat file(s) once, write single binary file
- Append new ASCII data to existing binary file
- Be sure to save original ASCII data (especially if using fst)
- Hardware can matter, CPU, GPU, etc
- Consider parallelization - R tools are available