High Performance R on Your Laptop, with an Introduction to Rcpp

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

https://meekj.github.io

7-November-2017 / UVa R Users

Overview

- I am not an expert, and other disclaimers...
- Range of native R performance: from horrible to HPC
- Concentrating on single-thread applications
  with a quick look at parallelization
- "Standard example" is 23 million observations of 5 variables, 813 MB
- Basic benchmarking techniques
- Extending and speeding-up R using C++ with Rcpp
- R version dependence
- Hardware - Desktop vs Laptop
- A real (simple) Rcpp application
- Providing access to a common C library
- Reading, and re-reading, large data sets

Three Ways to Increment a Vector with Base R - 1

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

> ## Use a for loop to increment every element
> t_start <- proc.time()    # Save current process times
>  for (i in 1:length(vec)) {
+      vec[i] <- vec[i] + 1
+  }
> proc.time() - t_start     # Get process time difference
   user  system elapsed     # Rough single run timing
  1.272   0.008   1.840
> 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
  1.272   0.008   1.840  for loop from method 1

> ## Do the loop another way, process time save not shown
> vec[1:length(vec)] <- vec[1:length(vec)] + 1
   user  system elapsed
  0.228   0.028   0.293
> 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 + 1
   user  system elapsed
  0.028   0.004   0.068      "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
  - 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(int Increment,
                                  NumericVector TheData) {
   int n = TheData.size();
   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(1, vec)
   user  system elapsed
  0.020   0.000   0.084
> str(vec)  num [1:23000000] 4 4 4 4 4 4 4 4 4 4 ...
```

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

Do Proper Benchmarking

Run each example 100 times and account for overhead

```
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] + 1
    }
)

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

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

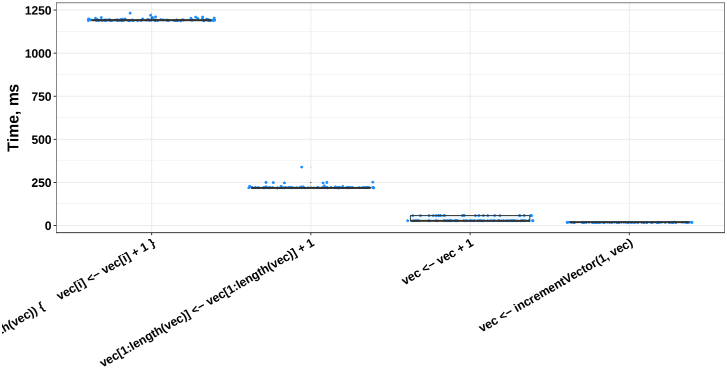
```
Unit: milliseconds
                                              expr        min         lq       mean     median         uq      max
for (i in 1:length(vec)) {vec[i] <- vec[i] + 1} 1187.43655 1189.21543 1193.26016 1190.90870 1193.85508 1232.245
   vec[1:length(vec)] <- vec[1:length(vec)] + 1  216.56800  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

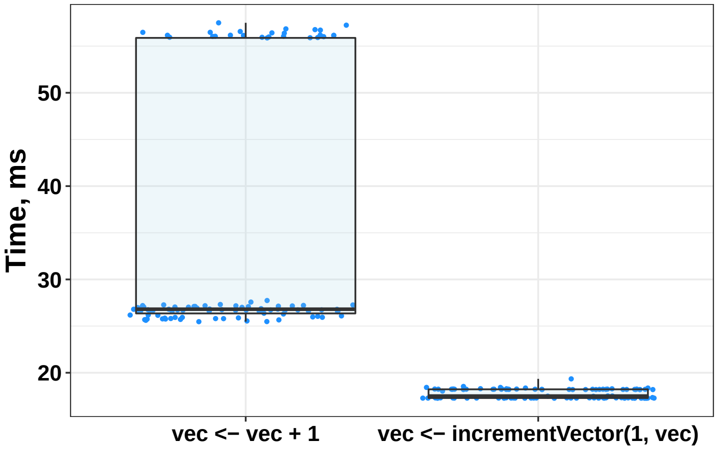
microbenchmark Notes

- 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
- Multiple expressions can be tested with adjustable order
- $expr contains the tested expression
- Individual measurements are in $time
- So, we can make boxplots

Incrementing Vector Elements - Four Ways



Incrementing Vector - Vectorized and Rcpp / C++



Does R Version Matter?

For the inefficient 'for' loop, YES !!!

```
: wx1:~ ; /usr/local/R-3.3.3/bin/R

R version 3.3.3 (2017-03-06) -- "Another Canoe"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

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

   user  system elapsed
 21.952   0.016  21.984
```

What?! 22 seconds? That took <1.3 seconds with 3.4.2 !

Changes between versions?

From the NEWS file:

```
CHANGES IN R 3.4.0:

 SIGNIFICANT USER-VISIBLE CHANGES:
...
* The JIT ('Just In Time') byte-code compiler is now enabled by
  default at its level 3. This means functions will be compiled on
  first or second use and top-level loops will be compiled and then
  run.  (Thanks to Tomas Kalibera for extensive work to make this
  possible.)
...
```

Test Byte Code Compilation in 3.3.3

The JIT byte-code compiler can be enabled manually in 3.3.x:

```
> library(compiler)
> enableJIT(3)

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

   user  system elapsed
  1.020   0.020   1.041
```

About 1 second, that's more like it!
Is that faster than 3.4.2 ?

Does R Version Matter Beyond Byte Compilation?

3.3.3 vs. 3.4.2 microbenchmark Results

```
Unit: milliseconds  100 evaluations
                                                expr
 for (i in 1:length(vec)) { vec[i] <- vec[i] + 1 }

 Ver        min       lq      mean   median       uq      max
 3.3.3  932.469  935.490  940.766  937.667  941.174  994.768
 3.4.2 1187.436 1189.215 1193.260 1190.908 1193.855 1232.245
```

A bit troublesome, but we will stick with 3.4.2
We avoid using 'for' in any case

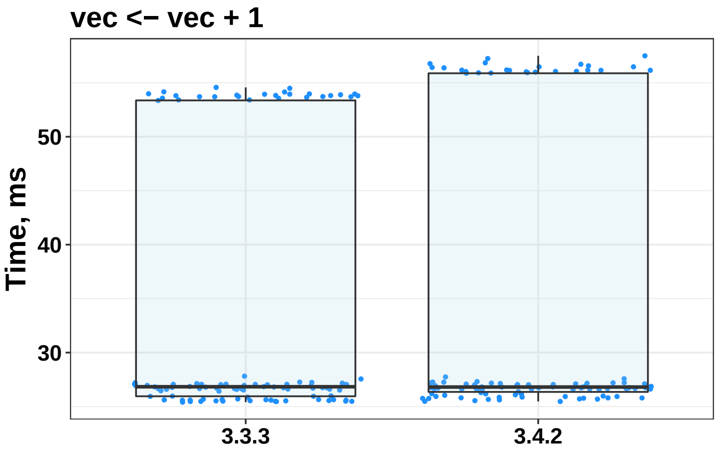
How about the vectorized method?

3.3.3 vs. 3.4.2 microbenchmark results
No need for byte compilation

```
Unit: milliseconds  100 evaluations
           expr
 vec <- vec + 1

 Ver        min      lq     mean   median       uq      max
 3.3.3  26.68542 50.9923 51.27475 51.25043 51.53207 56.73534
 3.4.2  28.49411 53.7712 54.27793 54.02434 54.47507 66.04011

"Quieter" system (fewer browser tabs open)
 3.3.3  25.4078  25.9512 33.54552 26.83606 53.38093  54.57069
 3.4.2  25.4744  26.3371 34.28377 26.80986 55.88667  57.49823
```

R version does not significantly affect performance for this example.

Vector Increment - R 3.3.3 vs R 3.4.2



How Much Does Hardware Matter?

Test systems, both 2015 vintage:

- Desktop: SuperMicro “SuperWorkstation”
  - Xeon E3-1276 v3 3.60GHz 4 core CPU
  - 32 GB ECC memory
  - Nvidia Quadro K620 GPU
  - Ubuntu 16.04.3 LTS with Xfce desktop environment
  - $1500.
- Laptop: MacBook Pro Early/Mid 2015
  - Core i7-4980HQ 2.80GHz 4 core CPU
  - 16 GB memory
  - Radeon GPU
  - MacOS Sierra 10.12.6 with XQuartz and MacPorts
  - $2500.

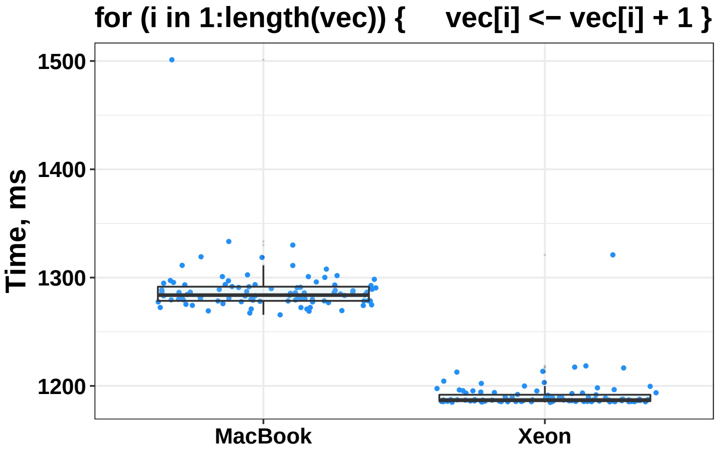
SSDs on both

Nearly identical CPU performance according to:

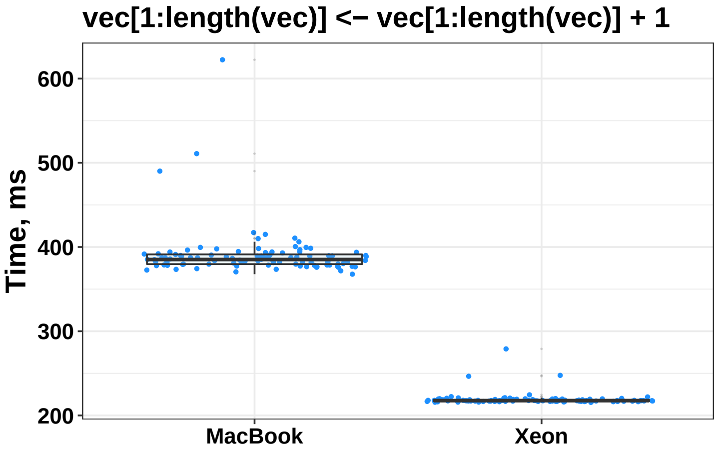
`http://www.cpubenchmark.net/singleThread.html`

R 3.4.2 / Emacs with ESS

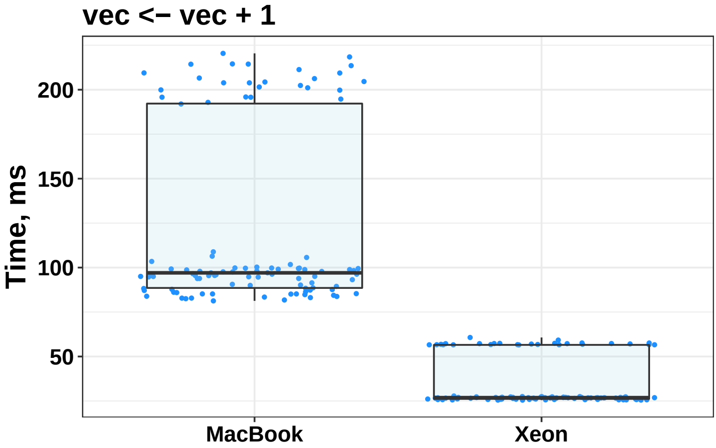
Pure R 'for' Loop - MacBook Pro vs Linux Xeon



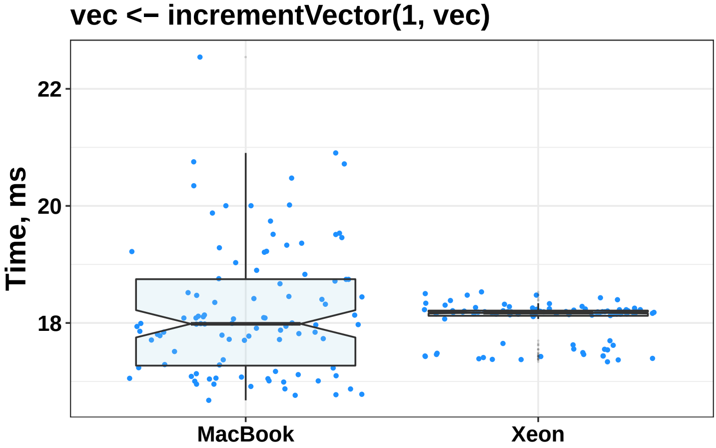
Pure R Loop 2 - MacBook Pro vs Linux Xeon



Pure R Vectorized - MacBook Pro vs Linux Xeon



C++ Loop - MacBook Pro vs Linux Xeon



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 a `for` loop: about 54 minutes
- Rcpp: (as low as) 780 milli-seconds !

Common Pure R Code - Read Data and Pre-process

```
library(dplyr)
library(readr)
library(lubridate)

## Read data
log_data <- read_delim(tf, delim = ' ', col_types = list(Time = col_datetime('%Y-%m-%dT%H:%M:%S')))

## Pre-process
log_data$Duration  <- log_data$Duration / 1000                             # milli-seconds to seconds
log_data$StartTime <- log_data$Time - ceiling(log_data$Duration) + 1 # Assume 1s resolution on log times

## Number of one second bins
MinTime <- min(log_data$StartTime)
timerange_s <- as.integer(difftime(max(log_data$Time), MinTime, units = 'sec')) + 1

## Index, 1 is first channel for R
log_data$StartSecond <- as.integer(difftime(log_data$StartTime, MinTime, units = 'sec'))

## Pre-allocate the result vector
ccu <- vector(mode = 'numeric', length = timerange_s)
```

Pure R Code - The Loop and Post-processing

```
## 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

        ## for (j in idx:k) {
        ##     ccu[j] <- ccu[j] + bytes_per_second       # An inner loop, how bad is it?
        ## }

    } 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;
}
```

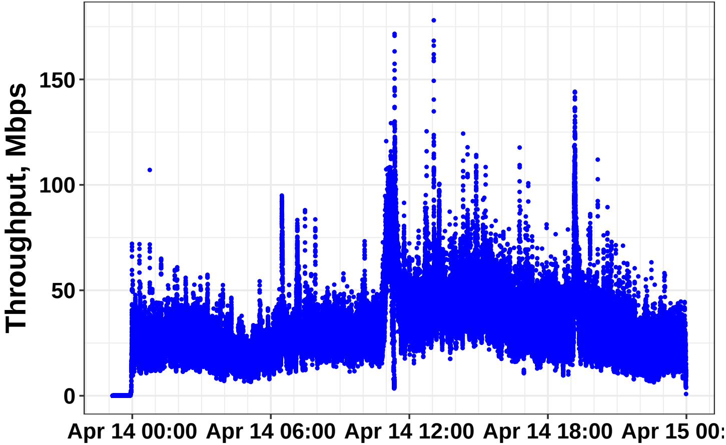
Hybrid R / C++ Code

```
library(tidyverse)
library(lubridate)
library(Rcpp)
sourceCpp("~/lab/Rpkgs/ConcurrentActivity/src/concurrent_activity.cpp")

tf <- '~/lab/R/data/as-20150414.dat'
log_data <- NULL
t_start_total <- proc.time()

log_data <- read_delim(tf, delim = ' ', col_types = list(Time = col_datetime('%Y-%m-%dT%H:%M:%S')))
t_end_read <- proc.time()

## log_data        <- log_data %>% filter(BytesRecv > 0) # Could drop events that will not add to throughput
log_data$Duration  <- log_data$Duration / 1000                      # milli-seconds to seconds
log_data$StartTime <- log_data$Time - ceiling(log_data$Duration) + 1 # Assume 1s resolution on log times
MinTime <- min(log_data$StartTime)

## Number of second bins
timerange_s       <- as.integer(difftime(max(log_data$Time), min(log_data$StartTime), units = 'sec')) + 1

## Pre-compute the index
log_data$StartSecond <- as.integer(difftime(log_data$StartTime, MinTime, units = 'sec'))
t_end_prep <- proc.time()

cca <- concurrentEstimatedThroughput(timerange_s, log_data$StartSecond, log_data$Duration, log_data$BytesRecv)

t_end_loop <- proc.time()

cca <- 8 * cca  / 1e6                                  # 8 bits / byte  - Mbps

cca_df      <- data.frame(ConcurrentVar = cca)         # Make the vector a data frame
cca_df$Time <- MinTime + seconds(seq(1:nrow(cca_df)) - 1) # Add time column
t_end_post  <- proc.time()
```

Run the Code

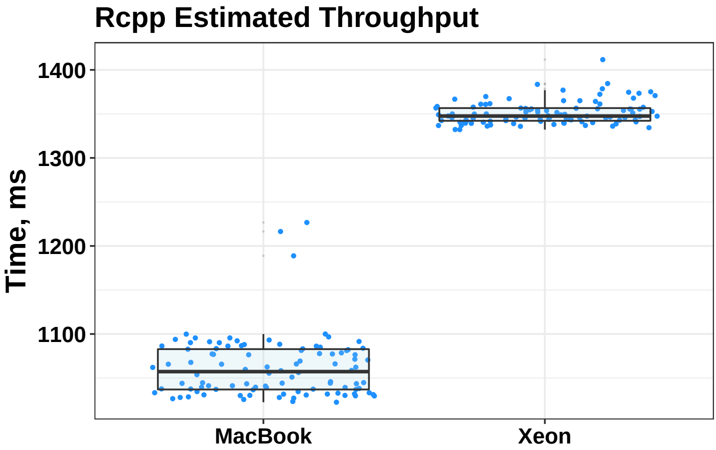
```
## Timing results
t_read  <- t_end_read - t_start_total
t_prep  <- t_end_prep - t_end_read
t_loop  <- t_end_loop - t_end_prep
t_post  <- t_end_post - t_end_loop
t_total <- t_end_post - t_start_total

> nrow(log_data)
[1] 22954489

> t_read
   user  system elapsed
 12.160   1.212  13.937
> t_prep
   user  system elapsed
  0.712   2.452   4.782
> t_loop
   user  system elapsed
  1.528   0.284   2.343
> t_post
   user  system elapsed
  0.136   0.000   0.180

> t_total
   user  system elapsed
 14.536   3.948  21.242
```

The Result - Estimated Throughput



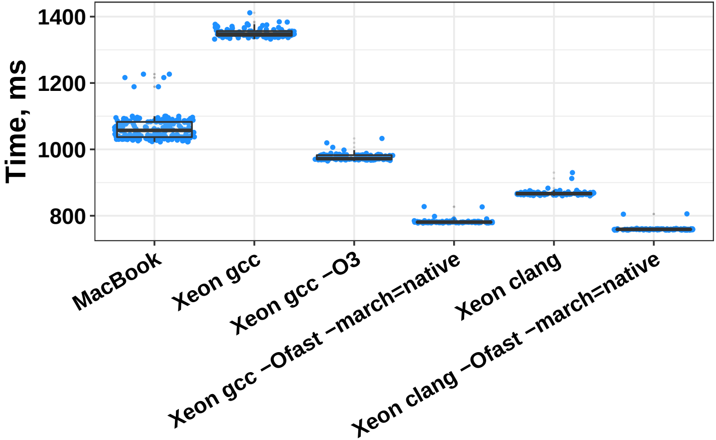
Estimated throughput for an Internet service over 24 hours with one second granularity.

MacBook vs “SuperWorkstation”, Again



Why is the MacBook significantly faster for Rcpp function?

MacBook vs "SuperWorkstation", Again



It's mostly the compiler optimization flags...

Selecting the Compiler & Flags

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

```
: wx1:~/.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++-3.8 -Qunused-arguments
CXXFLAGS += -Ofast -march=native
# CXXFLAGS += -O3
```

Compute Estimated Throughput

	Read	Prep	Loop	Post
Perl	98.3 s	9.6 s	6.6 minutes	0.30 s
Pure R	12.0 s	0.6 s	54 minutes	0.06 s
R / C++	12.0 s	0.6 s	0.78 seconds	0.06 s

23 million events
readr is used to read data in R
Read time includes date / time to seconds conversion
Perl post time includes writing result
A Python test would be interesting

Use Rcpp to Access C Library - libpcapR Package

- 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`
- Needs automated tests, vignette, etc
  and some users...
- Requires libpcap-dev package to be installed.
- Probably works only on Linux and Mac
- Does not yet appear on the list of GitHub R packages

Pre-made Rcpp Packages - Usually Performance Oriented

- dplyr and friends!
- RcppArmadillo - Armadillo Templated Linear Algebra Library
- RcppAnnoy - Annoy, a Library for Approximate Nearest Neighbors
- 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 → 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

Summary - So Far

- 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
- Use Rcpp where appropriate
  - Don't need to know a lot of C or C++
  - Be careful to not index past end of array, etc
  - 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

A Quick Look at Parallel Processing

- Use the doParallel package
  - Integrates foreach & parallel packages
  - Uses fork on UNIX-like systems
  - Uses snow (simple network of workstations) on Windows
  - Maintained by Microsoft (former Revolution Analytics team)
- Spins up N slave processes,
  where N may be number of cores or threads to use
- Can consume a lot of memory
- Data updates from master R process propagate
  (verify for your application)
- Can use a cluster of machines (using snow method),
  including mixed Windows & UNIX-like systems
- Consider number of cores and memory size when selecting hardware

```
> library(doParallel)
> vlength <- 23e7       # A 230 million point vector
> vec <- rnorm(vlength) # Fill with random numbers
> str(vec)   num [1:230000000] -2.754 -1.196 0.822 1.436 0.324 ...
> ## Simple (and fast) base R way
> singleTotal <- sum(atan(sqrt(vec * vec))) # A somewhat expensive operation
   user  system elapsed
  6.688   0.084   7.354

> ## Sum blocks in parallel
> NumBlocks  <- 4 # Number of data splits, can be larger
> CoresToUse <- 4
> cl <- makeCluster(CoresToUse)
> registerDoParallel(cl)

> BlockLength <- as.integer(length(vec) / NumBlocks)
> BlockCheck  <- length(vec) - NumBlocks * BlockLength
> if (BlockCheck) {cat("Choose an appropriate number of blocks!!\n")}
> ## Set up splitting indicies
> iseq <- seq(1,  length(vec), BlockLength)
> iseq <- as.integer(c(iseq, length(vec) + 1))
> str(iseq)   int [1:5] 1 57500001 115000001 172500001 230000001
> ## Sum blocks in parallel and then add results
> parallelTotal <- foreach(i=2:length(iseq), .combine=sum) %dopar% {
+     istart <- iseq[i-1]
+     iend   <- iseq[i] - 1
+     sum(atan(sqrt(vec[istart:iend] * vec[istart:iend])))
+ }
   user  system elapsed
  5.772   0.688  12.829
> singleTotal
[1] 134556974
> parallelTotal
[1] 134556974
> stopCluster(cl) # Shutdown slave processes
```

Summary - Parallelization

- A fair amount of overhead is required for setup
- Presumably best to use for computations requiring 1+ minutes
- Estimated throughput calculation is "embarrassingly parallel"
  Especially if data are in multiple files
- Do a validation check with a small data set
- Check out the High-Performance and Parallel Computing task view:
  `https://cran.r-project.org/web/views/HighPerformanceComputing.html`
- GNU Parallel can effectively run batch jobs across cores & machines

Reading, and Re-reading, Large Data Sets into R

- "Large" ASCII flat text files
- Many people will not consider these examples "large"
- But, if standard methods are used, it can require minutes to read multiple large files
- Very bad when running, or especially developing, batch jobs
- Example data set (same as above): 23 million observations of 5 variables, 813 MB
- Another example data set:
  - 366 daily files with per minute performance data for 30+ servers
  - 679 MB, 20.6 million observations of 5 variables
  - Smaller than today's example, but multiple file read took up to 16 minutes

Try Base R read.table

```
ascii_file <- '~/lab/R/data/as-20150414.dat'
log_data <- read.table(ascii_file, header = TRUE,
                                    stringsAsFactors = FALSE)
   user  system elapsed
 83.087   1.724  85.321 Mac
 88.630   1.723  91.085
 38.376   0.328  39.349 Xeon
 32.088   0.296  33.054

## Convert time string to POSIXct
log_data$Time <- as.POSIXct(log_data$Time,
                            format = "%Y-%m-%dT%H:%M:%S",
                            tz="UTC", origin="1970-01-01")
  11.508   0.416  12.717 Mac
   7.559   0.775   8.852
   9.840   0.124  10.748 Xeon
  10.204   0.084  10.851
```

Try Tidyverse readr

Note that read_delim does not handle variable width whitespace

```
library(readr)
log_data <- NULL

log_data <- read_delim(ascii_file, delim = ' ',
            col_types =
            list(Time = col_datetime('%Y-%m-%dT%H:%M:%S')))
```

```
##   user  system elapsed
## 10.842   1.420  15.905 Mac
## 10.173   0.700  12.215
## 11.992   0.232  12.861 Xeon
## 11.512   0.152  12.262
```

data.table's fread is Supposed to be Fast

```
library(data.table)

log_data <- fread(ascii_file)

   user  system elapsed
  3.728   0.072   4.406
  4.451   0.453   6.332
  3.804   0.096   4.466
  4.184   0.092   4.815
```

- Yes, it's pretty fast
- But, it overloads multiple dplyr and lubridate objects
- 4 seconds is still a long time when our computation takes 1 s
- Need to add as much as 10 s for time string conversion (faster methods exist)
- And, I don't routinely use data.table for anything else

How about other file formats?

- Native file formats
  - Write data frame with write.table (writes a new ASCII file)
  - Native binary .Rdata & .Rds formats
  - Loading .Rdata file uses original data frame name
  - readRDS will allow any data frame name to be loaded
  - RDS is best for general purpose use (in the Base R world)
- Other general purpose file formats
  - fst - "Lightning Fast Serialization of Data Frames for R"
  - Feather - Single format for R & Python

Write Data Frame to Base R Binary File

```
## Use caution when writing files, shell noclobber will not help

## Default compression
SaveRDSfile <- '~/lab/R/data/as-20150414.rds'
saveRDS(log_data, file=SaveRDSfile)

   user  system elapsed
 33.172   0.028  33.815


## No compression
SaveRDSfile2 <- '~/lab/R/data/as-20150414uc.rds'
saveRDS(log_data, file=SaveRDSfile2, compress = FALSE)

   user  system elapsed
  0.424   0.184   1.242
```

Write Data Frame to FST Binary File

```
library(fst)

## fst does not intrepret ~ (tilde) as home directory
FSTfile1 <- '/usr2/home/meekj/lab/R/data/as-20150414c0.fst'
write.fst(log_data, FSTfile1) # No compression
##    user  system elapsed
##   0.000   0.232   0.376

FSTfile2 <- '/usr2/home/meekj/lab/R/data/as-20150414c50.fst'
write.fst(log_data, FSTfile2, compress = 50) # 50% compression
##    user  system elapsed
##   0.180   0.140   0.407

FSTfile3 <- '/usr2/home/meekj/lab/R/data/as-20150414c100.fst'
write.fst(log_data, FSTfile3, compress = 100) # 100% compression
##    user  system elapsed
##   1.212   0.052   1.854
```

Read R Native Binary Format

```
> mb_readRDSfile1 <- microbenchmark(
+  tl <- readRDS(RDSfile1))

> mb_readRDSfile1
Unit: seconds
                    expr      min       lq     mean   median       uq      max neval
 tl <- readRDS(RDSfile1) 2.119655 2.125069 2.224458 2.133801 2.439007 2.484974   100


> mb_readRDSfile2 <- microbenchmark(
+   tl <- readRDS(RDSfile2))

> mb_readRDSfile2
Unit: milliseconds
                    expr      min       lq     mean   median       uq      max neval
 tl <- readRDS(RDSfile2) 550.4746 553.0599 611.0371 570.3467 610.9433 800.2684   100
```

Median times are in summary table below

Read FST Binary Format

FST does not preserve POSIXct type, use dplyr::mutate + as.POSIXct to fix

```
##  Uncompressed
> mb_readFSTfile1 <- microbenchmark({
+     tl <- read.fst(FSTfile1)
+     tl <- tl %>% mutate(Time = as.POSIXct(Time, tz="UTC", origin = "1970-01-01"))
+ })

##  50% compression level
> mb_readFSTfile2 <- microbenchmark({
+     tl <- read.fst(FSTfile2)
+     tl <- tl %>% mutate(Time = as.POSIXct(Time, tz="UTC", origin = "1970-01-01"))
+ })

## 100% compression level
> mb_readFSTfile3 <- microbenchmark({
+     tl <- read.fst(FSTfile3)
+     tl <- tl %>% mutate(Time = as.POSIXct(Time, tz="UTC", origin = "1970-01-01"))
+ })


> rbind(mb_readFSTfile1, mb_readFSTfile2, mb_readFSTfile3)
Unit: milliseconds
      min        lq      mean    median        uq      max neval
 309.2306  502.6339  631.2545  597.6924  788.9715 1072.811   100
 470.7730  660.8045  783.3025  736.3602  944.3054 1185.943   100
 812.3327 1040.9350 1151.0821 1103.5218 1293.5455 1586.537   100
```

Median times are in summary table below

File Read Performance Summary

23 Million Events of 5 Variables

Type	Method	Compression	Size, MB	Read Time, s
ASCII	read.table	NA	813	35 + 10
ASCII	data.table	NA	813	4 + 10
ASCII	readr	NA	813	12 + 0
Binary	RDS	TRUE	121	2.133
Binary	FST	100 %	106	1.103
Binary	FST	50 %	159	0.736
Binary	FST	0 %	526	0.597
Binary	RDS	FALSE	526	0.570

read.table & data.table require 10 s for string to POSIXct (using base function)
read.table took 80+ seconds on MacBook
readr's read_delim includes string to time conversion
RDS files contain the converted time in POSIXct format
FST does not preserve POSIXct type, conversion time from numeric is included (about 0.2 s)
Binary read times are the median value from 100 runs

Summary - Using Binary File Formats

- Read large ASCII flat file(s) once, write single binary file
  - Reading and parsing ASCII data is generally expensive
  - Reading multiple files is slower than reading a single large file
- Re-read data quickly as needed from binary file
- Append new data to existing binary file
- Be sure to save original ASCII data (especially if using fst)

Things we did not cover

- Code profiling - helpful for larger code blocks
- MRO - Microsoft R Open (formerly Revolution R)
  - Compiled with Intel Math Kernel Library (MKL)
  - YMMV, but it's usually a bit slower for me
  - It does give 8x improvement for large matrix multiplication
  - Installation caution: on Linux it changes the /usr/bin/R symlink
  - but does install in /opt/microsoft/ropen
  - MKL can be licensed free for personal / academic use, build your own R with it
- pqR - Pretty Quick R (compatible with R-2.15.1)
- Other special versions of R (see Advanced R book)
- Using GPUs
- "Real" Big Data: Hadoop, Spark, etc
- Making CRAN compatible Rcpp packages

Other Current R Related Projects

- Spectral analysis, peak finding and fitting, etc
- Lab equipment control & data acquisition:
  Oscilloscope, waveform generator, power supply, DMM, etc
- MACaddrR - Replace manufacturer portion of hardware address with abbreviated mfg name
- iperf network stress testing tools (Perl, C++, analysis in R)
- Rcpp log file parser to load data frames
- R version of Perl Net::Netmask to identify network that contains an IP address, will use Rcpp for bit manipulation
- Weather / water level analysis for Chesapeake Bay and Delaware River & Bay