Speeding up R

using C/C++/Fortran

Computing Club Presentation

Simon Fontaine

tinyurl.com/UMfastR

Student Seminar

Department of Statistics, University of Michigan

December 3rd, 2020

Disclaimer

- I am *not* a specialist on the matter
- Based on my experience as a user
- Please share your experiences!

Things to remember from this talk

1. Using compiled code can be **much** faster
2. Trade-off between **implementation** time and **execution** time
3. It’s **easier** than you might think

Background

Why are we using R?

- Interactive mode
- Rich libraries
- Syntax
- Plug-and-play
- Dynamic typing
- Fast
  - User time
  - Computing time (?)

```
> library(rpart)
> iris.tree <- rpart(Species ~ ., data=iris, method="class")
> iris.tree
n= 150

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
  2) Petal.Length< 2.45 50   0 setosa (1.00000000 0.00000000 0.00000
  3) Petal.Length>=2.45 100  50 versicolor (0.00000000 0.50000000 0.
    6) Petal.Width< 1.75 54   5 versicolor (0.00000000 0.90740741 0.
    7) Petal.Width>=1.75 46   1 virginica (0.00000000 0.02173913 0.9
```

Why is R *(somehow)* fast?

- Built on top of C
- Linear algebra operations use LAPACK (written in Fortran)
- Most packages use C/C++/Fortran for main calculations
    - R wrapper for convenient interface

```r
lm = function (formula, data, ...)
{
  # Prepare stuff
  # ...
  # Call main routine
  z = lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...)
  # Process output
  # ...
}
```

```r
lm.fit = function (x, y, ...)
{
  # Prepare stuff
  # ...
  # Call QR solver
  z <- .Call(C_Cdqrls, x, y, tol, FALSE)
  # Use QR to produce the OLS solution
}
```

```c
SEXP Cdqrls(SEXP x, SEXP y, SEXP tol, SEXP chk)
{
    /* Prepare stuff */
    /* ... */
    /* Call main Fortran77 routine */
    F77_CALL(dqrls)(REAL(qr), &n, &p, REAL(y), &ny, &rtol,
             REAL(coefficients), REAL(residuals), REAL(effects),
             &rank, INTEGER(pivot), REAL(qraux), work);
    /* Process output*/
    /* ... */
}
```

Example: the lm() function

Common packages using compiled code

C	C++	Fortran
base, stats, ...		base, stats, ...
data.table	RStan	randomForest
Matrix	e1071	quantreg
rpart	kernlab	gam
nnet	dplyr, tidyr	mvtnorm
survival	lme4	glmnet
splines	gbm	
mgcv	xgboost	

Prevalence of compiled code in R packages

	Count	% CRAN	% Compiled
CRAN Packages	16780	100.0%	
With compiled code	4071	24.3%	100.0%
Using Rcpp (C++)	2141	12.7%	56.6%
Other (C/Fortran)	1930	11.5%	43.4%

as of 12-02-2020

Why is R slow?

- Designed for usability, not performance
  - Interpreted, dynamically-typed, …
- Different code for same result have widely varying runtime
  - Non-trivial optimization

Performance, in Advanced R, 1st ed., Hadley Wickham http://adv-r.had.co.nz/Performance.html

Why bother?

- Implementation vs execution time
  - Spend more time on implementation to save on execution
- Repeated use of the same code
- Distribution of your code
  - Save other users’ time
  - Promote your work

Use case examples

- Solving optimization problems
- Iterative, recursive methods
- Sampling-based methods
- Complex data structures
- Evaluating a function many times
- ...

Typical structure: R wrapper function

1. [R] Process input
    a. Deal with different cases, missing/default input
    b. Some simple computations
    c. Prepare types and variables
2. [C/C++/Fortran] Call the main routine
3. [R] Process output
    a. Extract relevant variables
    b. Some simple computations
    c. Construct the R output

Examples

Example: Autocorrelation function

- Vector of length 10M
- First 1K lags

```
acf_r = function(x, lag.max=10){
    n = length(x)
    ac = rep(0, lag.max+1)
    ac[1] = 1
    mu = mean(x)
    sig = sd(x)
    xc = (x-mu)/sig
    for(k in seq(lag.max)){
        ac[k+1] = sum(xc[1:(n-k)]*xc[(k+1):n]) / (n-1)
    }
    return(ac)
}
```

Language	Time(s)	Relative
C (stats package)	11.85	1.08
C++	10.94	1.00
Fortran	11.06	1.02
R	129.60	11.85

Example: EM for Gaussian Mixture Model

- 1-dimensional
- Vector of length 1M
- 3 Components

```r
GMM.EM.R = function(x, k=2, eps=1.0e-8, max_iter=1000){
  init = GMM.init(x, k)
  mu = init$mu0; sig = init$sig0; pi = rep(1/k, k)
  n = length(x)
  p = matrix(0, n, k)
  llk = -Inf
  for(i in seq(max_iter)){
    # E STEP
    for(j in seq(k)){
      p[, j] = dnorm(x, mu[j], sig[j]) * pi[j]
    }
    # CHECK CONVERGENCE
    prev_llk = llk
    llk = mean(log(apply(p, 1, sum)))
    if(llk - prev_llk < eps) break
    p = sweep(p, 1, apply(p, 1, sum), "/")
    # M STEP
    pi = apply(p, 2, mean)
    mu = apply(sweep(p, 1, x, "*"), 2, sum) / apply(p, 2, sum)
    c = outer(x, mu, "-")
    sig2 = apply(p * c^2, 2, sum) / apply(p, 2, sum)
    sig = sqrt(sig2)
  }
  return(c(i, llk))
}
```

Language	Time(s)	Relative
C++	3.50	3.24
Fortran	1.08	1.00
R	76.2	70.56

Example: Cauchy density

- Vector of length 100M

```
dcauchy_r = function(x, loc, scale){
    return(scale / (pi * ((x-loc)^2 + scale^2)))
}
```

Language	Time(ms)	Relative
C (stats package)	1080	2.56
C++	422	1.00
Fortran	1250	2.96
R	717	1.70

Implementation

OLS using GD

Objective function

$$||Y-X\beta||^2/2n$$

Update

$$\beta=\beta+\eta X'(Y-X\beta)/n$$

JIT Rcpp

- Compile & use C++ code in a R Session
- Does not save the compiled code
- Good for simple or one-off uses
- Good for prototyping or development
- R functions:
  - Rcpp::evalCpp(string): evaluate a chunk of C++ code
  - Rcpp::cppFunction(string): compile and load a single C++ function
  - **Rcpp::sourceCpp(file): compile and load a .cpp file**

Fortran

- Implementation
- Compilation:
  - [Terminal] R CMD SHLIB file.f90
  - [R] system(“R CMD SHLIB file.f90”)
- Load:
  - [R] dyn.load("cauchy_f.so")
- Use:
  - .Fortran(“function”, args)

Rcpp package

- RStudio Project > Rcpp or RcppArmadillo
- Implementation
  - src/file.cpp
  - R/wrapper.R
- Build

Fortran package

- RStudio Project
- Implementation
  - src/file.f90
  - R/wrapper.R
- NAMESPACE
  - useDynLib(packageName)
- DESCRIPTION
  - NeedsCompilation: yes
- Build

Conclusion

Some guidelines

- Rcpp should be your first choice
  - Better interaction with R
  - Better support
  - Easier developpement workflow
  - Access to libraries
- Almost no reason to use pure C anymore
- Fortran can be slightly faster
  - When using only arithmetic/linear algebra
  - Clunkier workflow
  - Easier to learn, simpler syntax (?)
- Speed differences highly depend on your implementation
  - & compiler, architecture, etc.

References & further topics

Rcpp

- Extending R with C++: Motivation, Introduction and Examples - Part 1 by Dirk Eddelbuettel
  - [More advanced] Part 2
- Advanced R: Rewriting R code in C++ by Hadley Wickham
- Extending R with C++ by Dirk Eddelbuettel and James Joseph Balamuta
- RcppSugar: R-like behaviour in C++
- RcppEigen, RcppArmadillo: Linear algebra libraries

Fortran

- The Need for Speed Part 1: Building an R Package with Fortran (or C)
- The Need for Speed Part 2: C++ vs. Fortran vs. C
- Fortran and R – Speed Things Up
- Linking R and Fortran

R packages

- R packages by Hadley Wickham

Thank you!