As fast as it gets? Challenging (cpp) with julia

Philipp Adämmer Philipp Wittenberg

St Ralsund

11/2021

What is Julia?

- ► Compared to R, Julia¹ is a new programming language whose stable version has been released in 2018
- ▶ Julia is dynamically typed but also allows static type annotations
- ▶ Julia is fast and solves the two language problem
- ► The package environment is not as mature as R's but it is easy to call R or other languages
- ▶ Still suffers from compilation latency ("first time to plot")

Simple task

- ▶ Draw from $X \sim Bin(n = 100, p = 0.1)$
- ▶ Repeat the experiment $N=10^7$ times and compute coverage rates for the following 95% confidence intervals:
 - Wald
 - Wilson
 - Clopper-Pearson
- Compare the performance between R(cpp) and Julia
- ▶ Start with a naive loop implementation

Version I

Use package to compute CIs

```
function f1(n, p, N, \alpha)
2
      # Pre-allocate
3
4
      waldci = zeros(N, 2)
7
      # Loop
8
      Threads.@threads for jj = 1:N
9
10
11
       # Compute boolean vector
          randnrs = rand(n) .<= p
12
13
       # Compute CIs
14
                      = confint(BinomialTest(randnrs, p), level =
15
          waldtemp
                                 (1 - \alpha), method = :wald)
16
17
18
19
20
21
22
23
24
25
```

```
Version I
```

Use package to compute CIs

```
26
27
28
29
     # Fill matrices with Cls
30
        waldci[jj, 1] = waldtemp[1]
31
        waldci[jj, 2] = waldtemp[2]
32
33
34
35
36
37
38
39
40
     end
41
     # Compute coverage rates
42
       43
44
45
46
47
   end
```

Version I

Replace for loop

```
function f2(n, p, N, \alpha)
2
3
       # Compute boolean Matrix
4
          randnrs = rand(n, N) .<= p
5
6
7
       # Compute confidence intervals
          waldci
                    = ThreadsX.map(x -> confint(BinomialTest(x, p),
9
                      level=(1 - \alpha), method=:wald), eachcol(randnrs))
10
11
12
13
14
15
16
17
18
         # Compute coverage rates
19
20
21
22
23
24
     end
```

RCPP What is Rcpp³?

- ▶ R-package providing functions for seamless integration of R and C++
- ▶ Brings speed and performance to address R's bottlenecks
- Currently used by 2442 CRAN packages
- ▶ Include C++ libraries and other APIs via sourceCpp or cppFunction
- Can be used for parallel computations via RcppParallel²

²(Allaire et al. 2021;)

³(Eddelbuettel and François. 2011; J Stat Softw)

Confidence intervals

Wald interval

$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

Wilson interval

$$\frac{1}{1+\frac{z_{1-\alpha/2}^2}{n}} \Biggl(\widehat{p} + \frac{z_{1-\alpha/2}^2}{2n} \Biggr) \pm \frac{z_{1-\alpha/2}}{1+\frac{z_{1-\alpha/2}^2}{n}} \sqrt{\frac{\widehat{p}(1-\widehat{p})}{n} + \frac{z_{1-\alpha/2}^2}{4n^2}}$$

Clopper–Pearson interval

$$Beta\Big(\frac{\alpha}{2};x+1,n-x\Big) < \theta < Beta\Big(1-\frac{\alpha}{2};x+1,n-x\Big)$$

Rcpp Version I

confidence intervals from scratch

```
#include <Rcpp.h>
2
    using namespace Rcpp;
3
    // [[Rcpp::export]]
4
    NumericVector CIrcpp1(int n, double p, int N, double alpha) {
5
6
        const double& z = R::qnorm(a2, 0, 1, true, false), zz = z*z;
8
9
10
11
12
13
        for (int i = 0; i < N; i++) {
          x = R::rbinom(n, p);
14
15
          /* Wa.l.d. CT */
16
          c0 = x/n:
17
          c1 = z*sqrt(c0*(1-c0)/n);
18
          CIV(i, 0) = c0-c1;
19
          CIV(i, 1) = c0+c1;
20
```

Rcpp Version I

confidence intervals from scratch

```
/* Wilson CI */
21
22
          b1 = z/(1 + zz/n) * sqrt( (c0*(1-c0)/n) + zz/(4*n*n) );
23
24
25
26
27
          /* Clopper-Pearson CI */
          CIV(i, 4) = R::qbeta(a1, x, n-x+1, true, false);
28
29
30
31
        /* compute coverage rates */
32
        return NumericVector::create(
33
             mean( (CIV(_, 0) \le p) \& (p \le CIV(_, 1)) ),
34
35
36
        );
37
38
```

RCPP and RcppParallel

```
#include <Rcpp.h>
    #include <RcppParallel.h>
2
    // [[Rcpp::depends(RcppParallel)]]
3
    using namespace Rcpp;
    using namespace RcppParallel;
5
6
7
    struct CIworker : public Worker {
8
        /* input to read from */
9
10
11
12
13
        /* output matrix to write to */
14
15
16
        /* initialize */
17
18
         \rightarrow NumericMatrix mat) : X(X), a1(a1), a2(a2), z(z), n(n), mat(mat)
19
             /* calculate intervals and write to output matrix */
20
21
22
23
24
    };
```

RCPP and RcppParallel

```
// [[Rcpp::export]]
25
    NumericVector CIrcpp2(int n, double p, int N, double alpha) {
26
27
28
29
        /* draw vector of size N from Bin(n, p) */
30
        const NumericVector& X(Rcpp::rbinom(N, n, p));
31
32
        /* allocate the output matrix */
33
34
35
        /* create a worker */
36
        CIworker f1(X, a1, a2, z, n, CIV);
37
38
        /* call it with parallelFor */
39
        parallelFor(0, N, f1);
40
41
        /* return vector with coverage rates */
42
43
            mean( (CIV(_, 0) \le p) & (p \le CIV(_, 1)) ),
44
45
46
47
```

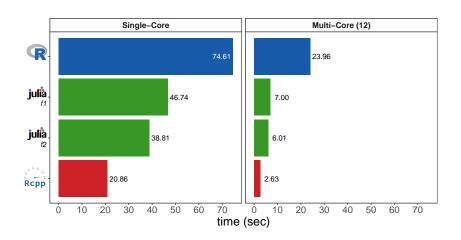
Benchmarking

Rcpp and Julia

```
# Load packages
 1
                                                        # Load packages
2
       library(Rcpp)
                                                          using HypothesisTests
3
                                                   3
4
                                                   4
5
                                                   5
6
     # Source scripts
                                                        # Include scripts
                                                   6
7
       sourceCpp("singlecore.cpp")
                                                          include("Code1.il")
                                                    7
8
       sourceCpp("multicore.cpp")
                                                          include("Code2.jl")
9
                                                   9
10
     # Set parameters
                                                        # Set paramters
                                                   10
11
                                                   11
12
                                                   12
13
                                                   13
14
                                                   14
15
                                                   15
16
                                                        # Benchmark functions
                                                   16
17
     # Benchmark functions
                                                   17
                                                         Obenchmark f1(n, p, N, \alpha) seconds = 70
18
       microbenchmark(
                                                   18
                                                         Obenchmark f2(n, p, N, \alpha) seconds = 70
19
        CIrcpp1(n, p, N, alpha),
20
        CIrcpp2(n, p, N, alpha),
21
       times = 10)
```

Benchmark results

single and multi-core



Custom function to compute CIs

```
function ciallf(x::Int64, z::Float64, n::Int64,
2
                  a1::Float64, a2::Float64)
3
4
      # Wald CI
5
        c0 = x/n
        c1 = z*sqrt(c0*(1 - c0)/n)
7
8
9
      # Wilson CI
10
        b0 = (1/(1 + z^2/n))*(c0 + (z^2)/(2*n))
11
        b1 = (z/(1 + (z^2/n)))*sart((c0*(1 - c0)/n) + z^2/(4*n^2))
12
13
14
15
      # Clopper-Pearson CI
        cplow = (x == 0 ? 0.0 : quantile(Beta(x, n - x + 1), a1))
16
        cpup = (x == n ? 1.0 : quantile(Beta(x + 1, n - x), a2))
17
18
19
      # Return CIs
20
        [(c0 - c1) (c0 + c1) (b0 - b1) (b0 + b1) cplow cpup]
21
22
23
     end
```

Version II

Use custom function and Distributions.jl⁴ to draw from Bin(n, p)

```
function f3(n::Int64, p::Float64, N::Int64, α::Float64)
2
3
4
       # Compute quantile
         z = quantile(Normal(), 1 - \alpha/2)
6
7
       # Draw # of successes
         dist = Binomial(n, p)
9
         randnrs = rand(dist, N)
10
11
12
       # Compute CIs
13
         ciall = ThreadsX.map(x -> ciallf(x, z, n, \alpha/2, (1 - \alpha/2)),
14
                 randnrs)::Vector{Matrix{Float64}}
15
16
17
       # Compute coverage rates
18
19
20
21
22
23
     end
```

⁴(Besançon et al. 2021; J Stat Softw)

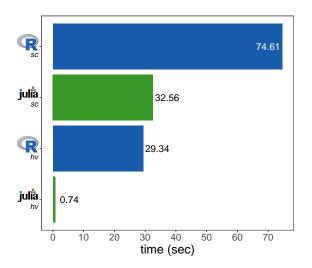


Use Static Arrays and draw CP before loop

```
function ciallf(x::Int64, z::Float64, n::Int64,
                     cplow::Vector{Float64}, cpup::Vector{Float64},
2
3
                    p::Float64)
4
    # Wa.l.d. CT
       c0 = x/n:
       c1 = z*sqrt(c0*(1 - c0)/n)
8
9
10
     # Wilson CI
       b0 = (1/(1 + z^2/n))*(c0 + (z^2)/(2*n))
11
       b1 = (z/(1 + (z^2/n)))*sqrt((c0*(1 - c0)/n) + z^2/(4*n^2))
12
13
14
     # Clopper-Pearson CIs already computed
15
16
                          # --- #
17
     # Return CIs
18
19
       SVector{3, Bool}((c0 - c1) <= p <= (c0 + c1),
                         (b0 - b1) <= p <= (b0 + b1),
20
                         cplow[x + 1]  <= p  <= cpup[x + 1])
21
22
23
    end
```

Benchmark results

Julia vs. R



sc: single core & full CI calculation; hv: single core & half vectorized

Rcpp fast version

unordered map

```
#include <Rcpp.h>
1
    using namespace Rcpp;
2
    // [[Rcpp::plugins(cpp11)]]
3
4
    // [[Rcpp::export]]
5
    NumericVector CIrcpp3(int n, double p, int N, double alpha) {
6
7
9
10
11
      std::unordered_map<int, double> CPL, CPU;
12
13
14
15
16
      /* unordered map for Clopper-Pearson CI values */
17
      for (int j = 0; j \le n; j++) {
18
        CPL[j] = R::qbeta(a1, j, n-j+1, true, false);
19
20
21
```

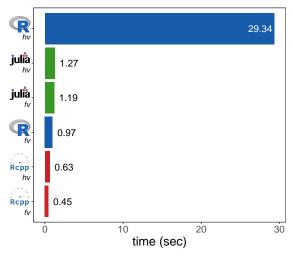
Rcpp fast version

unordered map

```
22
      /* get intervals */
      for (i = X.begin(); i != X.end(); ++i) {
23
24
        /* Wald CI */
25
26
27
28
29
30
       /* Wilson CI */
31
     b0 = (1/(1 + zz/n)) * (c0 + zz/(2*n));
32
        b1 = z/(1 + zz/n) * sqrt( (c0*(1-c0)/n) + zz/(4*n*n) );
33
34
35
36
37
38
        if ((CPL[*i] <= p) && (p <= CPU[*i])) L3++;
39
40
41
      /* compute coverage rates */
42
      return NumericVector::create(L1/N, L2/N, L3/N);
43
44
```

Benchmarks

using on-board functions



hv: half vectorized

fv: full vectorized

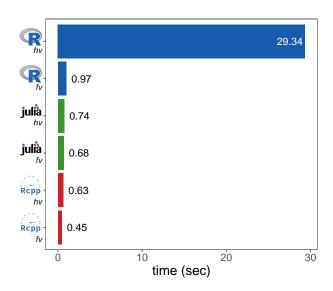
fast version

Use RCall

```
function f5(n::Int64, p::Float64, N::Int64, \alpha::Float64)
2
11
        # Use R to draw from Bin(n, p)
12
          randnrs::Vector{Int64} = (rcopy(Vector{Int64}, R"rbinom($N, $n, $p)"))
13
14
15
16
18
19
20
21
22
23
    end
```

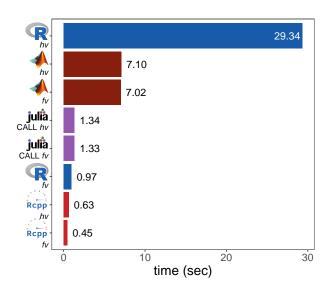
Benchmarks

using rbinom()



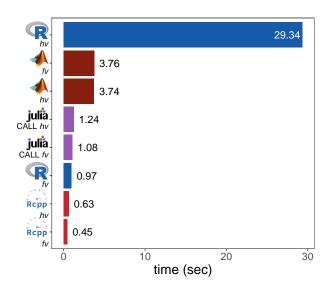
Additional benchmarks

using on-board functions



Additional benchmarks

using rbinom()



Summary







- ▶ Don't use loops in R for such a task, vectorize!
- ▶ Julia and Rcpp do not benefit much from vectorization
- Optimize your code before using multiple threads!
- Rccp forces you to write type stable functions. Writing type stable functions in Julia can be tricky sometimes
- Using R within Julia is easier and faster than using Julia within R
- Using Rcpp is (slightly) faster for this task than using Julia(Call)

Thank you for your attention

References

- Bezanson, J. et al. (2017). "Julia: A fresh approach to numerical computing". SIAM review 59.1, pp. 65–98
- Allaire, J. et al. (2021). RcppParallel: Parallel Programming Tools for 'Rcpp'. R package version 5.1.4. URL: https://CRAN.R-project.org/package=RcppParallel
- Eddelbuettel, D. and R. François (2011). "Rcpp: Seamless R and C++ Integration". J Stat Softw 40.8, pp. 1–18
- Besançon, M. et al. (2021). "Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem". *J Stat Softw* 98.16, pp. 1–30