As fast as it gets?

Challenging R(cpp) with julia

Philipp Adämmer Philipp Wittenberg

StRalsund
11/2021

What is Julia?

- Compared to R, Julia[1] 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")

[1] (Bezanson et al. 2017; *SIAM review*)

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, α)

 # Pre-allocate
  waldci   = zeros(N, 2)
  wilsonci = zeros(N, 2)
  cpci     = zeros(N, 2)

 # Loop
  Threads.@threads for jj = 1:N

  # Compute boolean vector
     randnrs = rand(n) .<= p

  # Compute CIs
     waldtemp    = confint(BinomialTest(randnrs, p), level =
                           (1 - α), method = :wald)

     wilsontemp  = confint(BinomialTest(randnrs, p), level =
                           (1 - α),  method = :wilson)

     cptemp      = confint(BinomialTest(randnrs, p), level =
                           (1 - α),  method = :clopper_pearson)
                                .
                                .
                                .
```

Version I

Use package to compute CIs

```
26                                          .
27                                          .
28                                          .
29
30      # Fill matrices with CIs
31         waldci[jj, 1]     = waldtemp[1]
32         waldci[jj, 2]     = waldtemp[2]
33
34         wilsonci[jj, 1]   = wilsontemp[1]
35         wilsonci[jj, 2]   = wilsontemp[2]
36
37         cpci[jj, 1]       = cptemp[1]
38         cpci[jj, 2]       = cptemp[2]
39
40      end
41
42      # Compute coverage rates
43        [ sum(waldci[:, 1]    .<= p .<= waldci[:, 2])/N
44          sum(wilsonci[:, 1] .<= p .<= wilsonci[:, 2])/N
45          sum(cpci[:, 1]     .<= p .<= cpci[:, 2])/N      ]
46
47   end
```

Version I

Replace for loop

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

What is Rcpp[3]?

- 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 RcppParallel[2]

[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

$$\frac{1}{1+\frac{z^2_{1-\alpha/2}}{n}}\left(\hat{p}+\frac{z^2_{1-\alpha/2}}{2n}\right) \pm \frac{z_{1-\alpha/2}}{1+\frac{z^2_{1-\alpha/2}}{n}}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}+\frac{z^2_{1-\alpha/2}}{4n^2}}$$

- Clopper–Pearson interval

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

Rcpp Version I

confidence intervals from scratch

```
1   #include <Rcpp.h>
2   using namespace Rcpp;
3
4   // [[Rcpp::export]]
5   NumericVector CIrcpp1(int n, double p, int N, double alpha) {
6
7       const double& a1 = alpha/2, a2 = 1-alpha/2;
8       const double& z  = R::qnorm(a2, 0, 1, true, false), zz = z*z;
9
10      double x, c0, c1, b0, b1;
11      NumericMatrix CIV(N, 6);
12
13      for (int i = 0; i < N; i++) {
14        x = R::rbinom(n, p);
15
16        /* Wald CI */
17        c0 = x/n;
18        c1 = z*sqrt(c0*(1-c0)/n);
19        CIV(i, 0) = c0-c1;
20        CIV(i, 1) = c0+c1;
```

Version I

confidence intervals from scratch

```
21          /* Wilson CI */
22          b0 = (1/(1 + zz/n)) * (c0 + zz/(2*n));
23          b1 = z/(1 + zz/n) * sqrt( (c0*(1-c0)/n) + zz/(4*n*n) );
24          CIV(i, 2) = b0 - b1;
25          CIV(i, 3) = b0 + b1;
26
27          /* Clopper-Pearson CI */
28          CIV(i, 4) = R::qbeta(a1, x, n-x+1, true, false);
29          CIV(i, 5) = R::qbeta(a2, x+1, n-x, true, false);
30      }
31
32      /* compute coverage rates */
33      return NumericVector::create(
34          mean( (CIV(_, 0) <= p) & (p <= CIV(_, 1)) ),
35          mean( (CIV(_, 2) <= p) & (p <= CIV(_, 3)) ),
36          mean( (CIV(_, 4) <= p) & (p <= CIV(_, 5)) )
37      );
38  }
```

Rcpp and RcppParallel

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

Rcpp and RcppParallel

```
25  // [[Rcpp::export]]
26  NumericVector CIrcpp2(int n, double p, int N, double alpha) {
27
28      const double& a1 = alpha/2, a2 = 1-alpha/2;
29      const double&  z = R::qnorm(a2, 0, 1, true, false);
30      /* draw vector of size N from Bin(n, p) */
31      const NumericVector& X(Rcpp::rbinom(N, n, p));
32
33      /* allocate the output matrix */
34      NumericMatrix CIV(N, 6);
35
36      /* create a worker */
37      CIworker f1(X, a1, a2, z, n, CIV);
38
39      /* call it with parallelFor */
40      parallelFor(0, N, f1);
41
42      /* return vector with coverage rates */
43      return NumericVector::create(
44          mean( (CIV(_, 0) <= p) & (p <= CIV(_, 1)) ),
45          mean( (CIV(_, 2) <= p) & (p <= CIV(_, 3)) ),
46          mean( (CIV(_, 4) <= p) & (p <= CIV(_, 5)) ));
47  }
```

Benchmarking

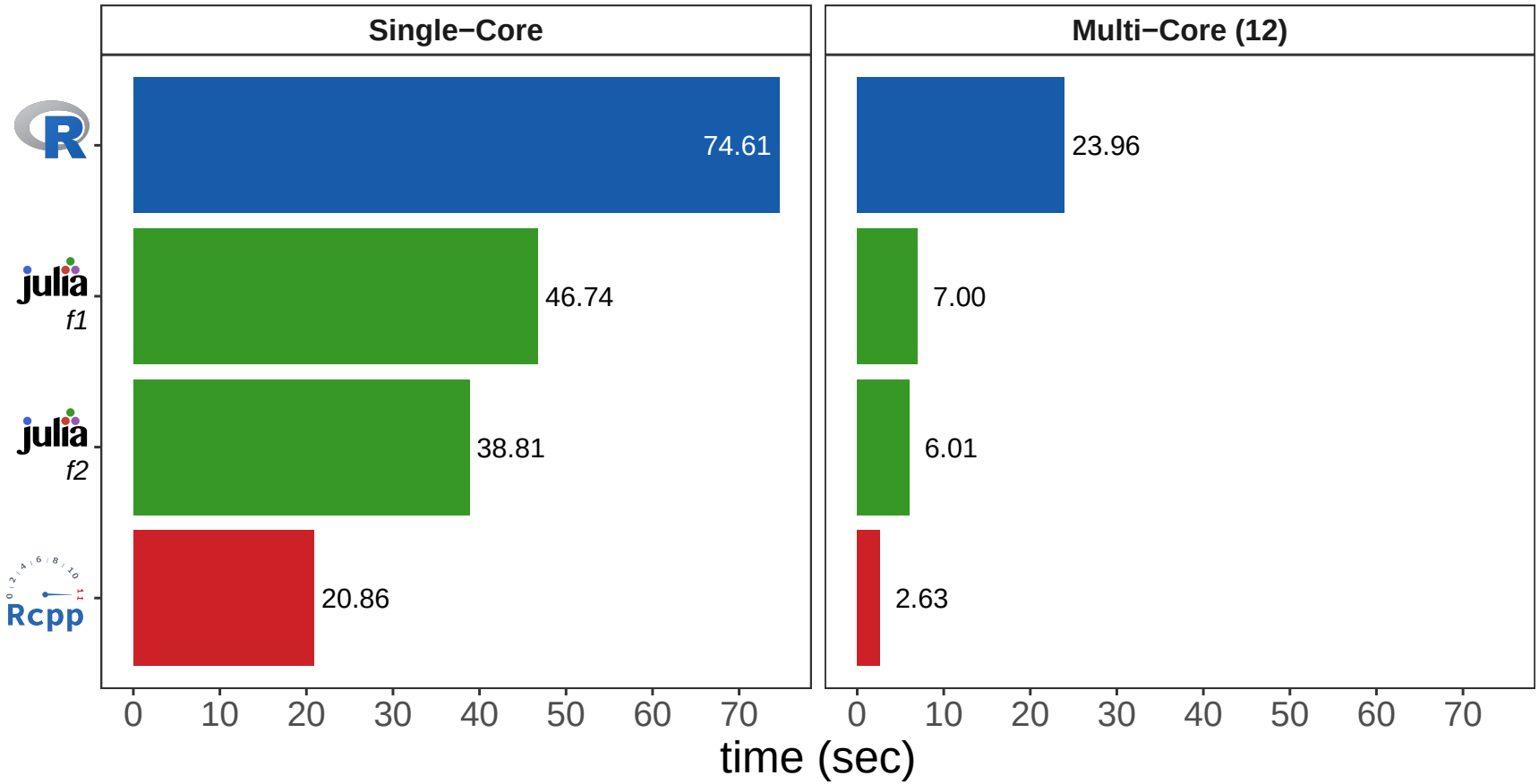
Rcpp and Julia

```r
# Load packages
  library(Rcpp)
  library(RcppParallel)
  library(microbenchmark)

# Source scripts
  sourceCpp("singlecore.cpp")
  sourceCpp("multicore.cpp")

# Set parameters
  n     <- 100
  p     <- .1
  N     <- 1e7
  alpha <- .05
  setThreadOptions(numThreads = 12)

# Benchmark functions
  microbenchmark(
   CIrcpp1(n, p, N, alpha),
   CIrcpp2(n, p, N, alpha),
  times = 10)
```

```julia
# Load packages
  using HypothesisTests
  using ThreadsX
  using BenchmarkTools

# Include scripts
  include("Code1.jl")
  include("Code2.jl")

# Set paramters
  n = 100;
  p = 0.1;
  N = 10^7;
  α = 0.05;

# Benchmark functions
 @benchmark f1($n, $p, $N, $α) seconds = 70
 @benchmark f2($n, $p, $N, $α) seconds = 70
```

Benchmark results

single and multi-core



Version II

Custom function to compute CIs

```
function ciallf(x::Int64, z::Float64, n::Int64,
                a1::Float64, a2::Float64)


  # Wald CI
    c0   = x/n
    c1   = z*sqrt(c0*(1 - c0)/n)


  # Wilson CI
    b0    = (1/(1 +  z^2/n))*(c0 + (z^2)/(2*n))
    b1    =  (z/(1 + (z^2/n)))*sqrt((c0*(1 - c0)/n) + z^2/(4*n^2))


  # Clopper-Pearson CI
    cplow = (x == 0 ? 0.0 : quantile(Beta(x,     n - x + 1), a1))
    cpup  = (x == n ? 1.0 : quantile(Beta(x + 1, n - x),     a2))


  # Return CIs
    [(c0 - c1) (c0 + c1) (b0 - b1) (b0 + b1) cplow cpup]

 end
```

Version II

Use custom function and Distributions.jl[4] to draw from Bin(n, p)

```
1   function f3(n::Int64, p::Float64, N::Int64, α::Float64)
2
3
4     # Compute quantile
5       z  = quantile(Normal(), 1 - α/2)
6
7
8     # Draw # of successes
9       dist    = Binomial(n, p)
10      randnrs = rand(dist, N)
11
12
13    # Compute CIs
14      ciall = ThreadsX.map(x -> ciallf(x, z, n,  α/2, (1 - α/2)),
15              randnrs)::Vector{Matrix{Float64}}
16
17
18    # Compute coverage rates
19    [ sum(getindex.(ciall, 1) .<= p .<= getindex.(ciall, 2))/N
20      sum(getindex.(ciall, 3) .<= p .<= getindex.(ciall, 4))/N
21      sum(getindex.(ciall, 5) .<= p .<= getindex.(ciall, 6))/N  ]
22
23   end
```

[4] (Besançon et al. 2021; *J Stat Softw*)

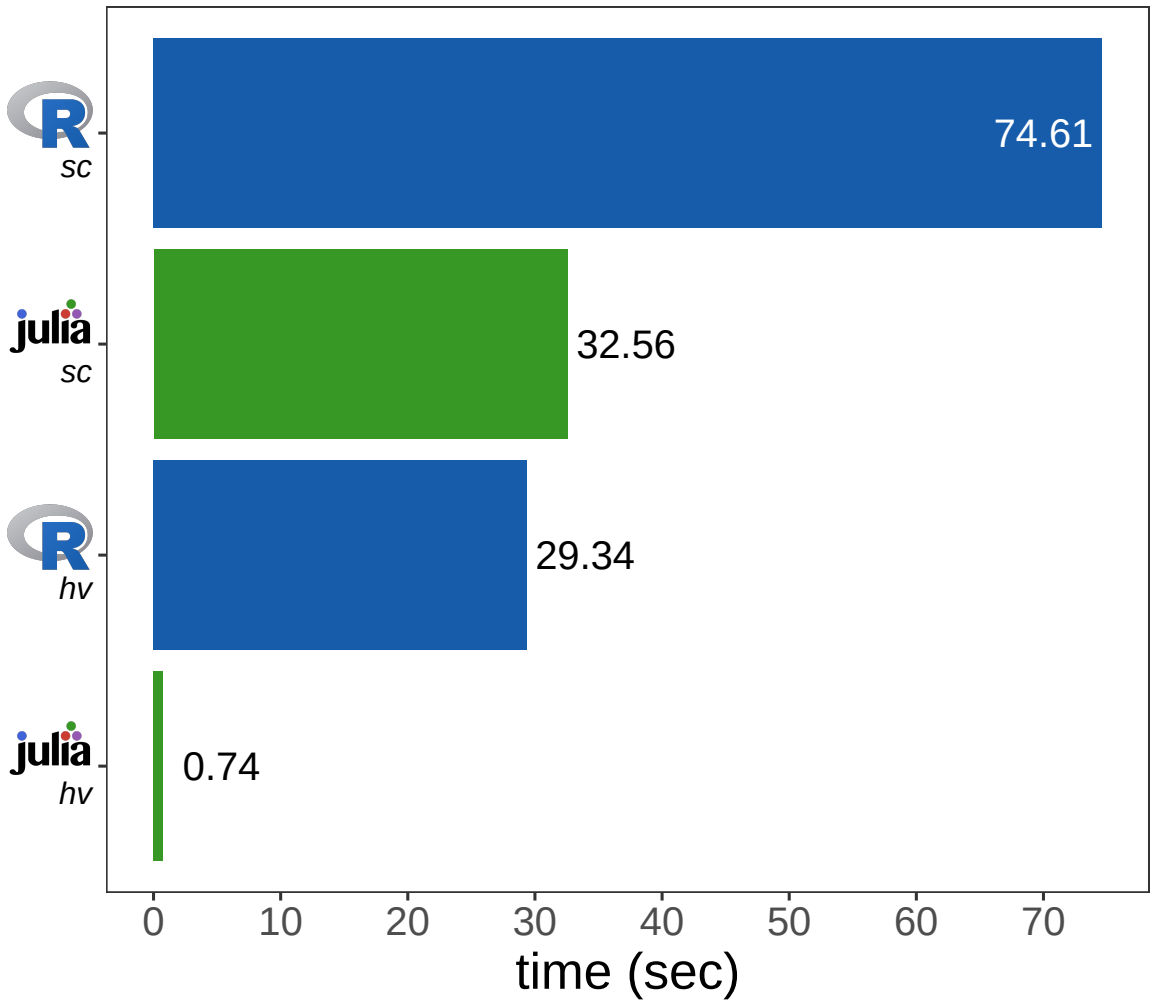
Version III

Use Static Arrays and draw CP before loop

```
function ciallf(x::Int64, z::Float64, n::Int64,
                cplow::Vector{Float64}, cpup::Vector{Float64},
                p::Float64)

# Wald CI
   c0   = x/n;
   c1   = z*sqrt(c0*(1 - c0)/n)


 # Wilson CI
   b0    = (1/(1 +  z^2/n))*(c0 + (z^2)/(2*n))
   b1    = (z/(1 + (z^2/n)))*sqrt((c0*(1 - c0)/n) + z^2/(4*n^2))


 # Clopper-Pearson CIs already computed
                      # --- #

 # Return CIs
   SVector{3, Bool}((c0 - c1)     <= p <= (c0 + c1),
                    (b0 - b1)     <= p <= (b0 + b1),
                    cplow[x + 1]  <= p <=  cpup[x + 1])

end
```

Benchmark results

Julia vs. R



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

Rcpp fast version

unordered map

```
1   #include <Rcpp.h>
2   using namespace Rcpp;
3   // [[Rcpp::plugins(cpp11)]]
4
5   // [[Rcpp::export]]
6   NumericVector CIrcpp3(int n, double p, int N, double alpha) {
7
8     double c0, c1, b0, b1;
9     const double& a1 = alpha/2, a2 = 1-alpha/2;
10    const double& z  = R::qnorm(a2, 0, 1, true, false), zz = z*z;
11    double WaL, WaU, WiL, WiU, L1, L2, L3;
12    std::unordered_map<int, double> CPL, CPU;
13
14    NumericVector X(Rcpp::rbinom(N, n, p));
15    NumericVector::iterator i;
16
17    /* unordered map for Clopper-Pearson CI values */
18    for (int j = 0; j <= n; j++) {
19      CPL[j] = R::qbeta(a1, j, n-j+1, true, false);
20      CPU[j] = R::qbeta(a2, j+1, n-j, true, false);
21    }
```

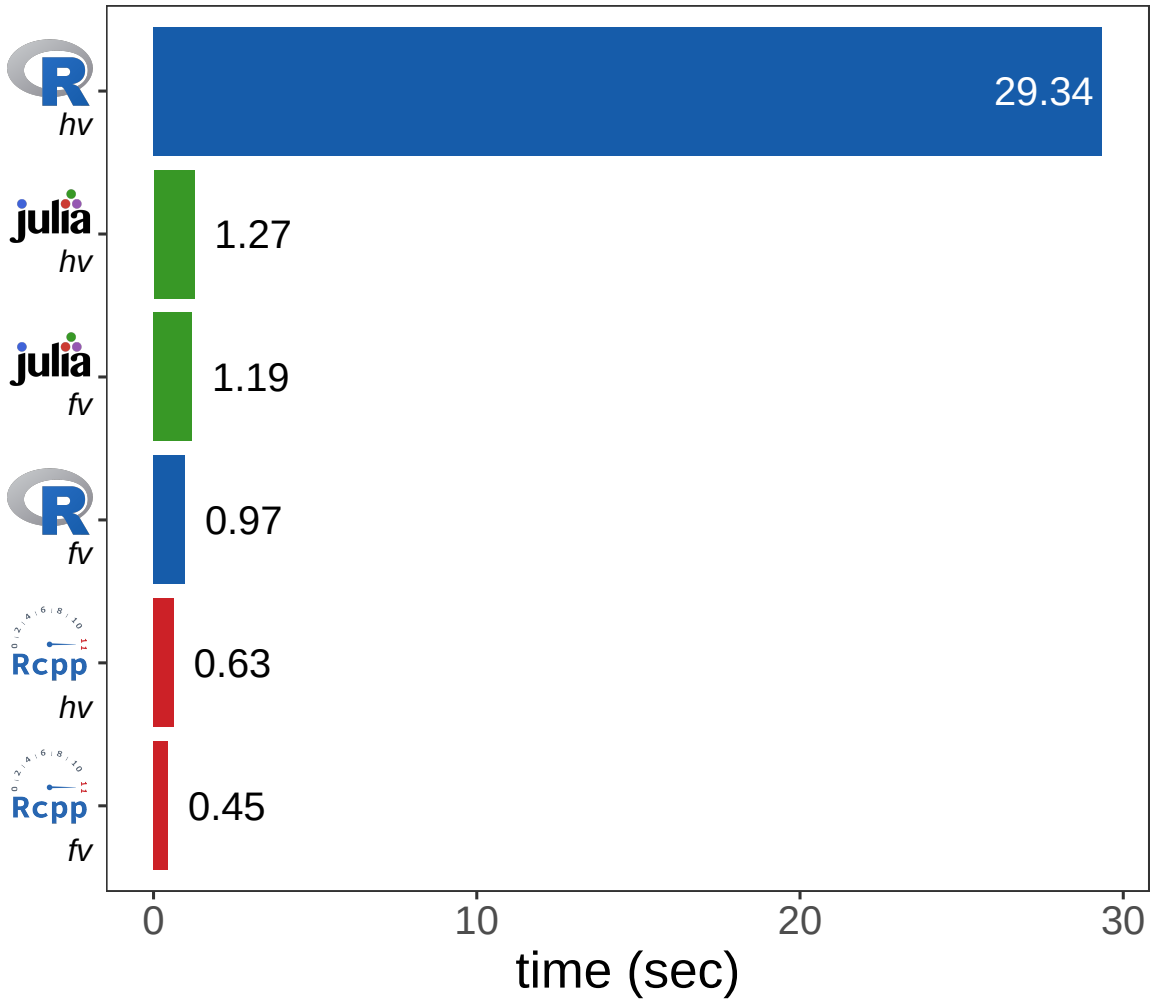
Rcpp fast version

unordered map

```
22      /* get intervals */
23      for (i = X.begin(); i != X.end(); ++i) {
24
25        /* Wald CI */
26        c0  = *i/n;
27        c1  = z*sqrt(c0*(1-c0)/n);
28        WaL = c0-c1;
29        WaU = c0+c1;
30
31        /* Wilson CI */
32        b0 = (1/(1 + zz/n)) * (c0 + zz/(2*n));
33        b1 = z/(1 + zz/n) * sqrt( (c0*(1-c0)/n) + zz/(4*n*n) );
34        WiL = b0-b1;
35        WiU = b0+b1;
36
37        if (    (WaL <= p) && (p <= WaU))     L1++;
38        if (    (WiL <= p) && (p <= WiU))     L2++;
39        if ((CPL[*i] <= p) && (p <= CPU[*i])) L3++;
40      }
41
42      /* compute coverage rates */
43      return NumericVector::create(L1/N, L2/N, L3/N);
44    }
```

Benchmarks

using on-board functions



hv: half vectorized fv: full vectorized

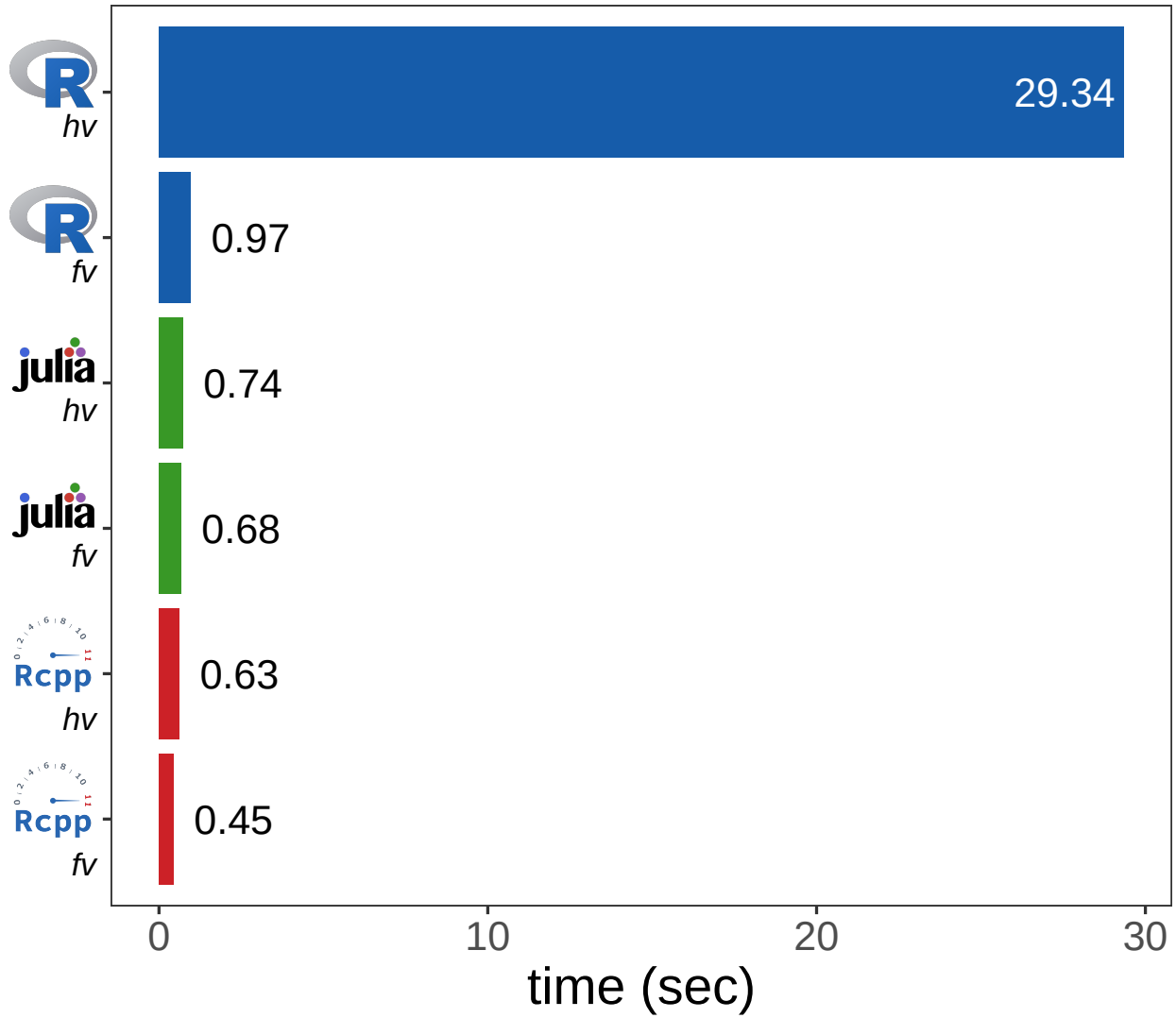
fast version

Use RCall

```
1   function f5(n::Int64, p::Float64, N::Int64, α::Float64)
2
3
4
5                                   .
6                                   .
7                                   .
8
9
10
11
12      # Use R to draw from Bin(n, p)
13        randnrs::Vector{Int64} = (rcopy(Vector{Int64}, R"rbinom($N, $n, $p)"))
14
15
16
17                                  .
18                                  .
19                                  .
20
21
22
23  end
```

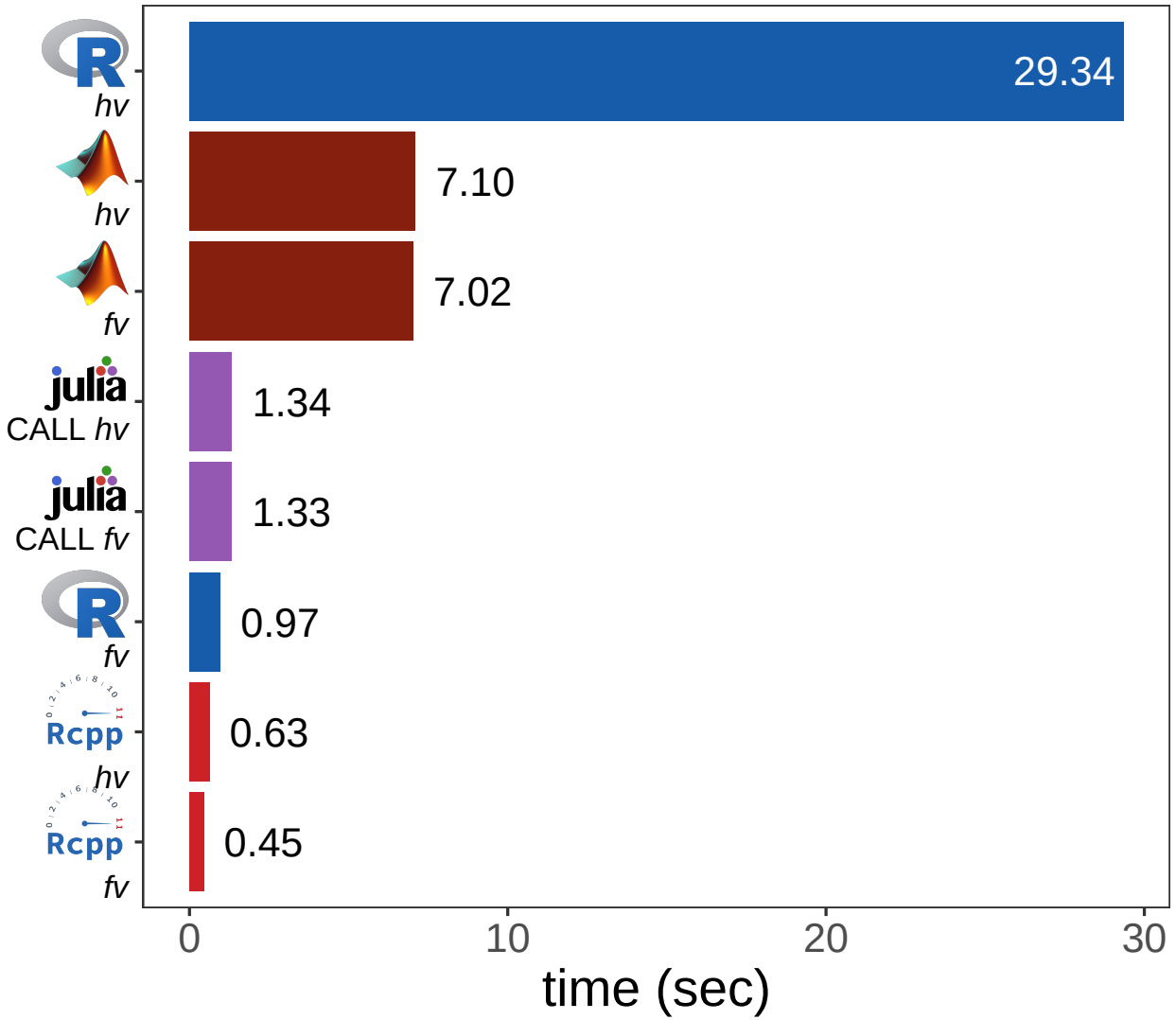
Benchmarks

using rbinom()



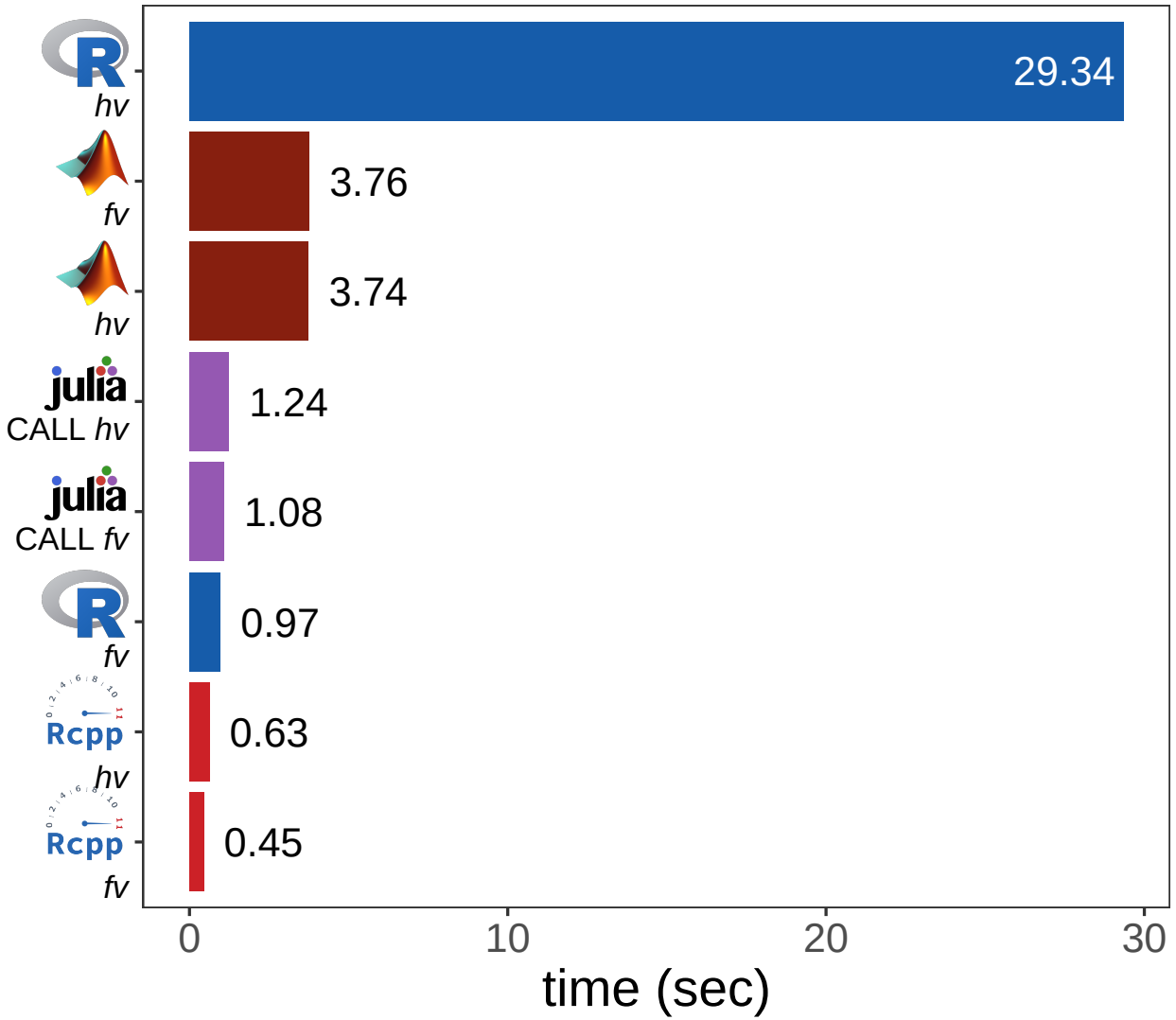
Additional benchmarks

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!
- Rcpp 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