Sharing your method with R(cpp)

Four guidelines for faster code

Byron C Jaeger

Wake Forest University School of Medicine

2022/08/08 (updated: 2022-08-04)

What do biostatisticians do?

1. Create methods that can engage with contemporary data and make valid conclusions

2. Share those methods

3. - ??? many other things...

Graduate training

- Number of required courses on statistics and probability: 4+
- Number of required courses on software development: ???

So how do most of us learn?

r/ProgrammerHumor

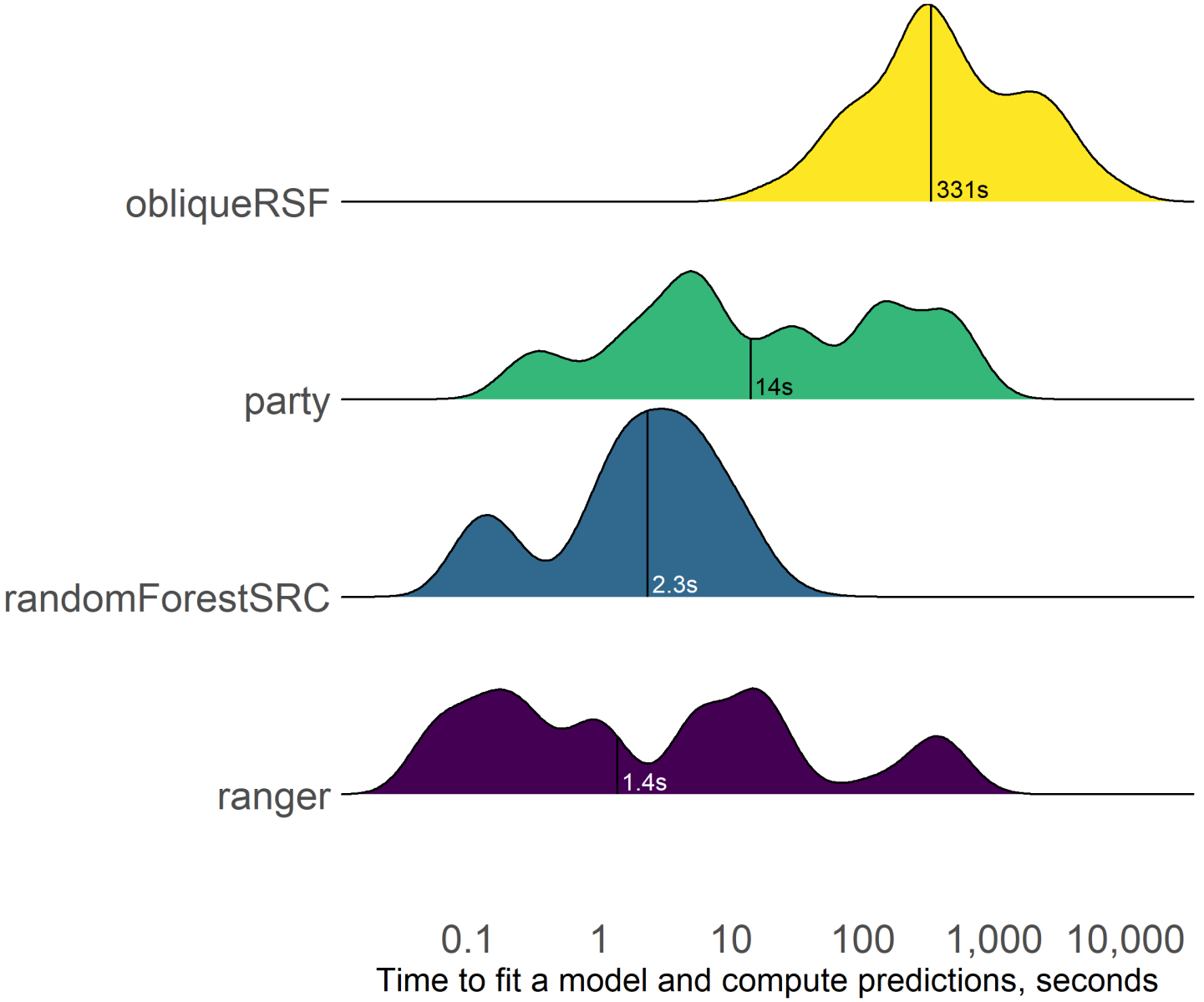
obliqueRSF

In 2019, I made the `obliqueRSF` R package

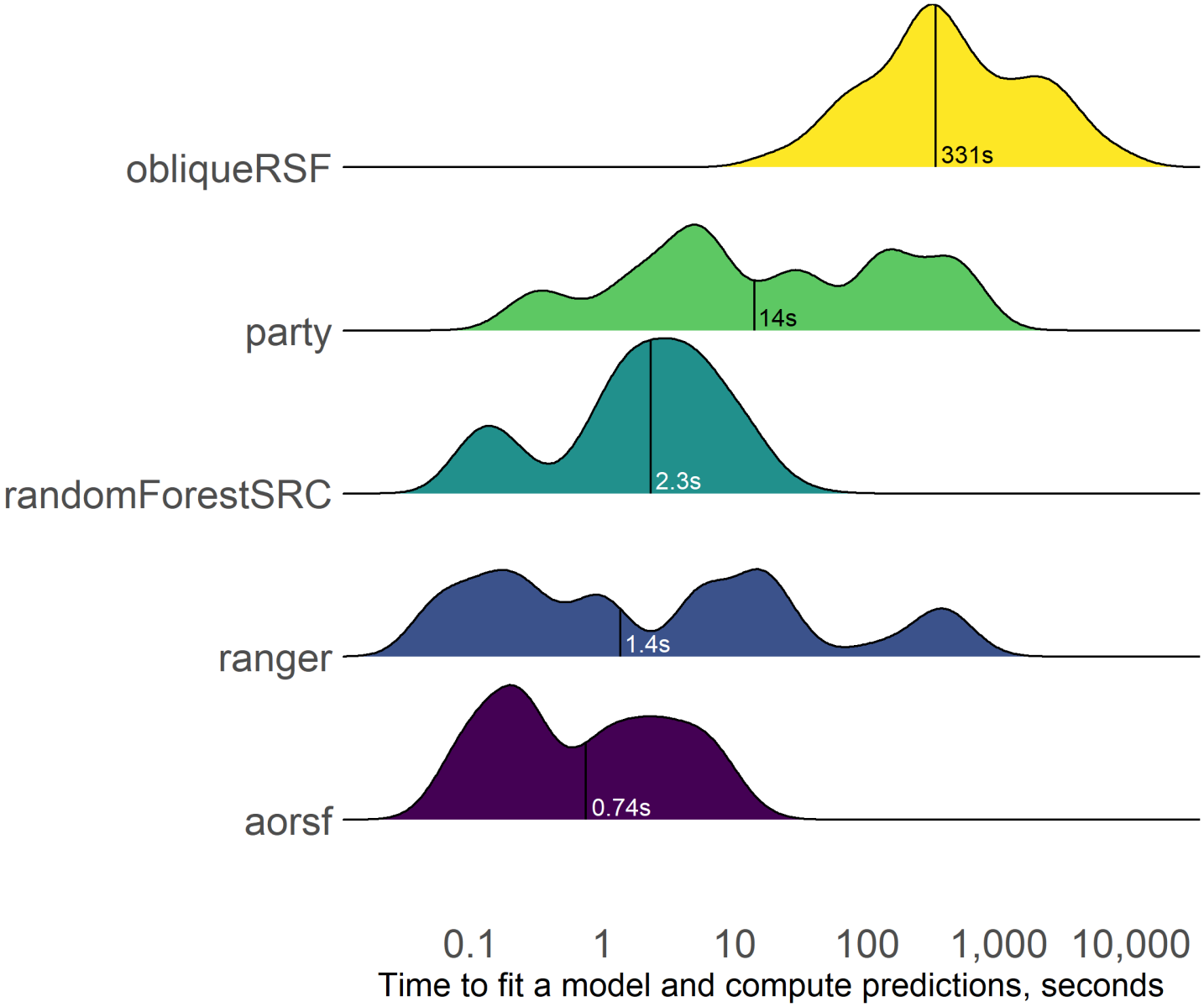
- oblique random survival forests

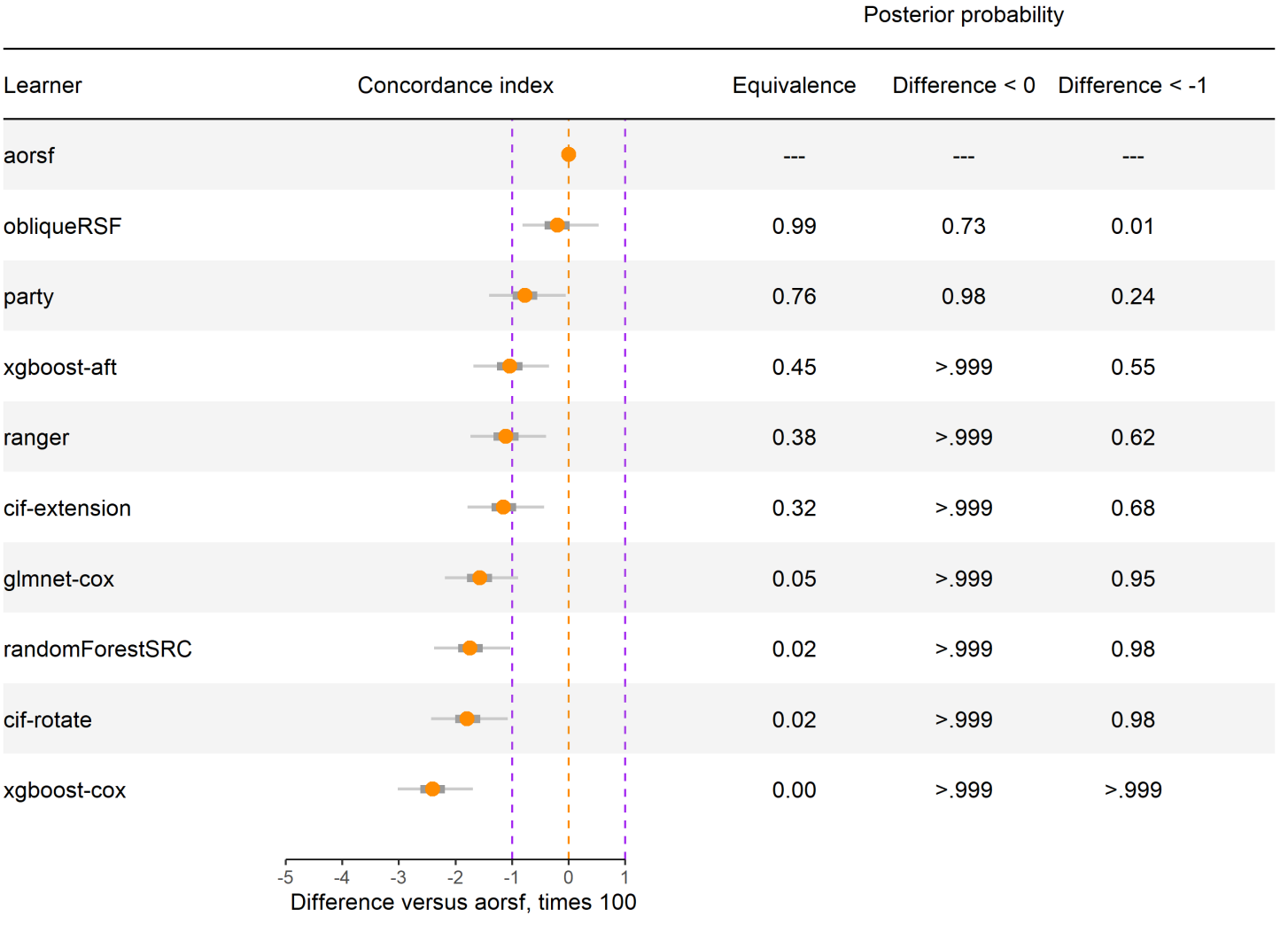
`obliqueRSF` had higher prediction accuracy versus:

- `randomForestSRC`,
- `ranger`,
- `party`.
- but was also hundreds of times slower.

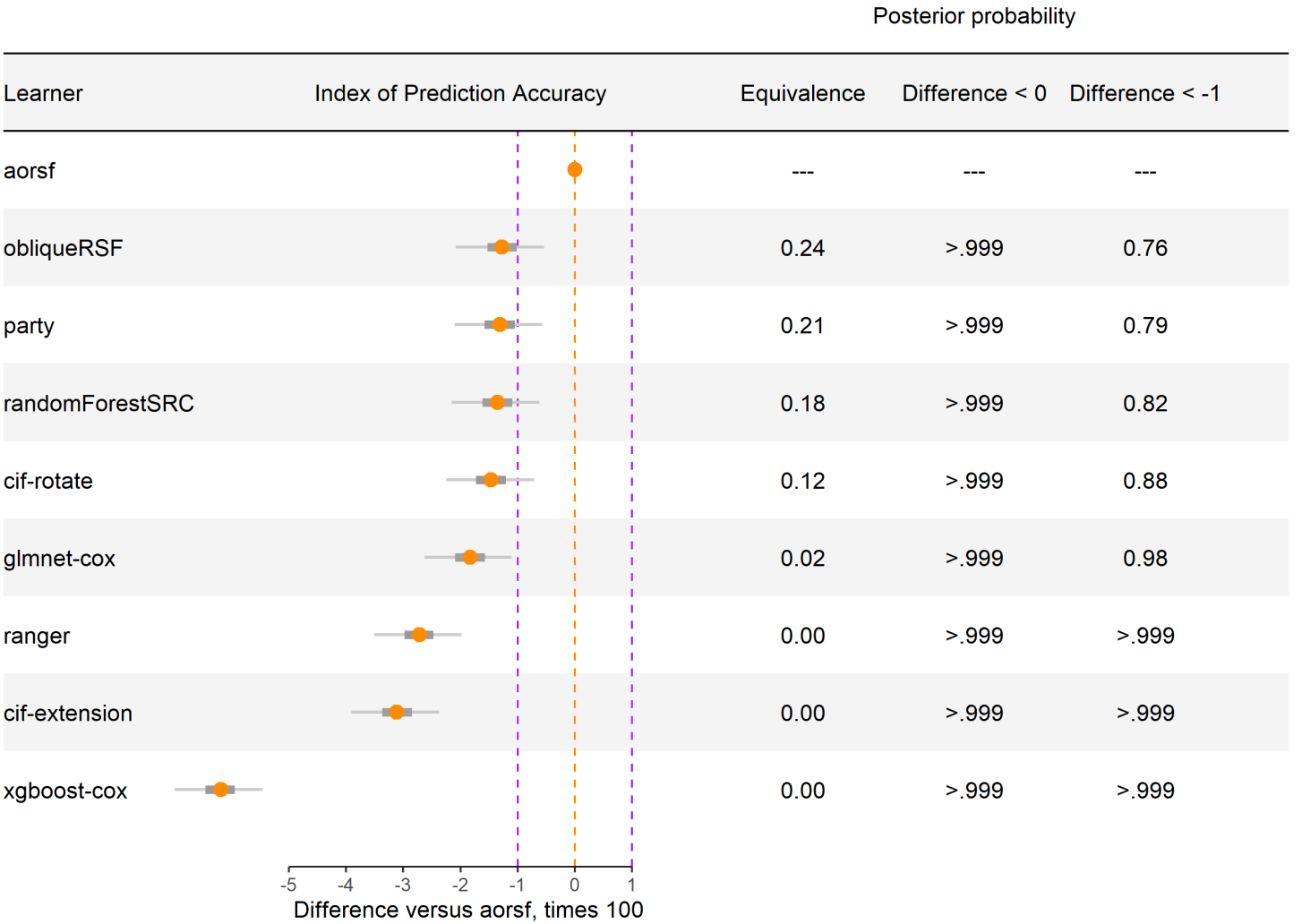

obliqueRSF
331s
party
14s
randomForestSRC
2.3s
ranger
1.4s
0.1
1
10
100
1,000
10,000
Time to fit a model and compute predictions, seconds


Tell me the truth...I'm...I'm ready to hear it.
NOBODY USES SLOW R PACKAGES
imgflip.com


obliqueRSF
331s
party
14s
randomForestSRC
2.3s
ranger
1.4s
aorsf
0.74s
0.1
1
10
100
1,000
10,000
Time to fit a model and compute predictions, seconds


Posterior probability
Learner
Concordance index
Equivalence
Difference < 0
Difference < -1
aorsf

---
obliqueRSF
0.99
0.73
0.01
party
0.76
0.98
0.24
xgboost-aft
0.45
>.999
0.55
ranger
0.38
>.999
0.62
cif-extension
0.32
>.999
0.68
glmnet-cox
0.05
>.999
0.95
randomForestSRC
0.02
>.999
0.98
cif-rotate
0.02
>.999
0.98
xgboost-cox
0.00
>.999
>.999
-5
-4
-3
-2
-1
0
1
Difference versus aorsf, times 100


Posterior probability
Learner
Index of Prediction Accuracy
Equivalence
Difference < 0
Difference < -1
aorsf

---
obliqueRSF
0.24
>.999
0.76
party
0.21
>.999
0.79
randomForestSRC
0.18
>.999
0.82
cif-rotate
0.12
>.999
0.88
glmnet-cox
0.02
>.999
0.98
ranger
0.00
>.999
>.999
cif-extension
0.00
>.999
>.999
xgboost-cox
0.00
>.999
>.999
-5
-4
-3
-2
-1
0
1
Difference versus aorsf, times 100

aorsf paper (https://arxiv.org/abs/2208.01129)

[Submitted on 1 Aug 2022]

Accelerated and interpretable oblique random survival forests

Byron C. Jaeger, Sawyer Welden, Kristin Lenoir, Jaime L. Speiser, Matthew W. Segar, Ambarish Pandey, Nicholas M. Pajewski

The oblique random survival forest (RSF) is an ensemble supervised learning method for right-censored outcomes. Trees in the oblique RSF are grown using linear combinations of predictors to create branches, whereas in the standard RSF, a single predictor is used. Oblique RSF ensembles often have higher prediction accuracy than standard RSF ensembles. However, assessing all possible linear combinations of predictors induces significant computational overhead that limits applications to large-scale data sets. In addition, few methods have been developed for interpretation of oblique RSF ensembles, and they remain more difficult to interpret compared to their axis-based counterparts. We introduce a method to increase computational efficiency of the oblique RSF and a method to estimate importance of individual predictor variables with the oblique RSF. Our strategy to reduce computational overhead makes use of Newton-Raphson scoring, a classical optimization technique that we apply to the Cox partial likelihood function


ASRF

Four guidelines for faster code

1. Always be benchmarking

Why benchmark?

- it gives you data on how fast your code runs
- it encourages you to write more modular functions (a good thing)

How to benchmark?

Step 1. Define a task: Count events per group (status of 1 $\Rightarrow$ event, 3 groups)

```
# status = c(1, 0, 1, ...)
status = sample(x = c(0L, 1L), size = 1e5, replace = TRUE)

# group = c(2, 0, 1, ...)
group = sample(x = c(0L, 1L, 2L), size = 1e5, replace = TRUE)
```

Step 2. Define the reference competitors

```
# competitors: table() and tapply()
table(status, group)[2, ]
```

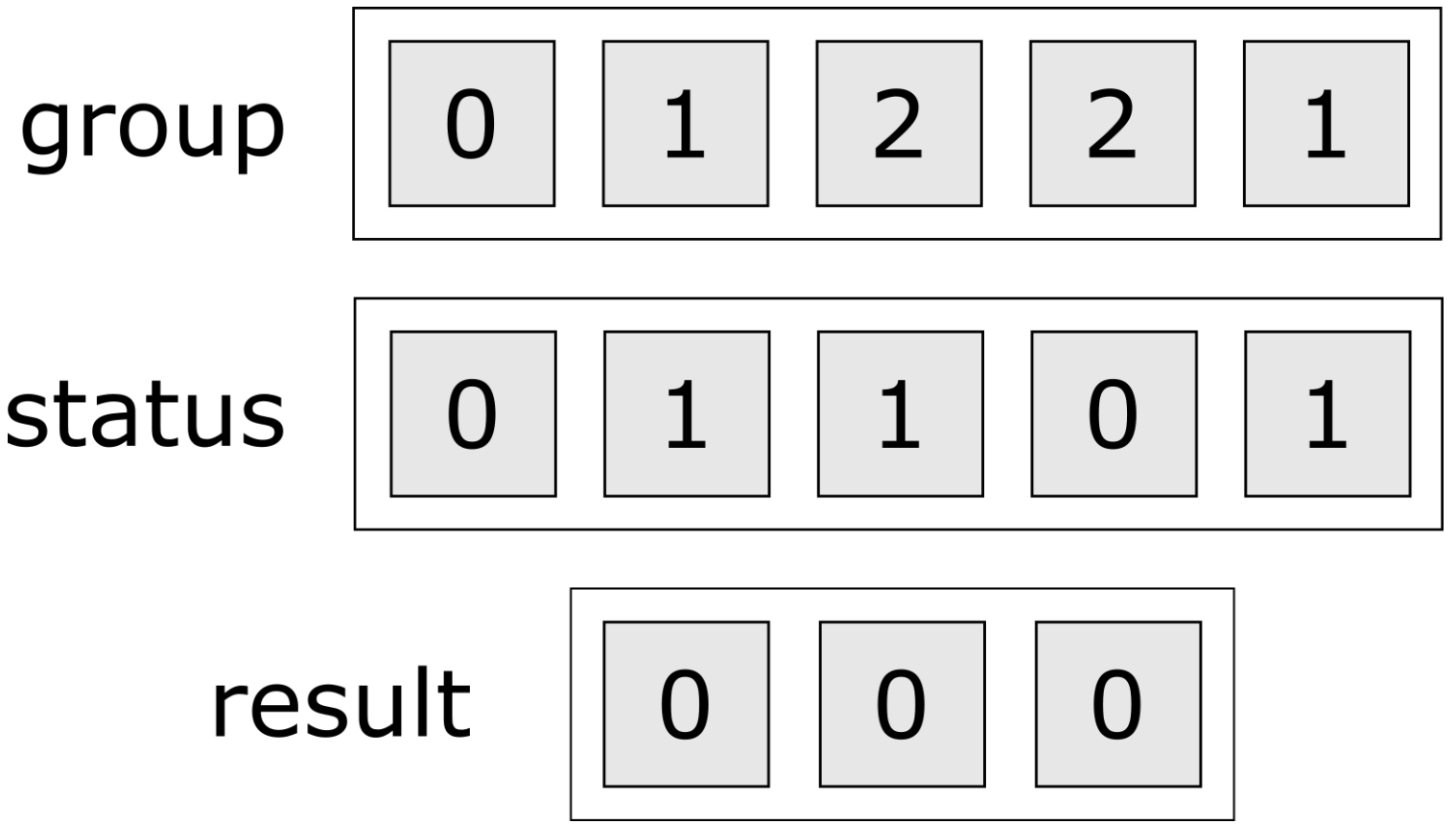
```
##     0     1     2
## 16537 16560 16865
```

```
tapply(status, group, FUN = sum)
```

```
##     0     1     2
## 16537 16560 16865
```

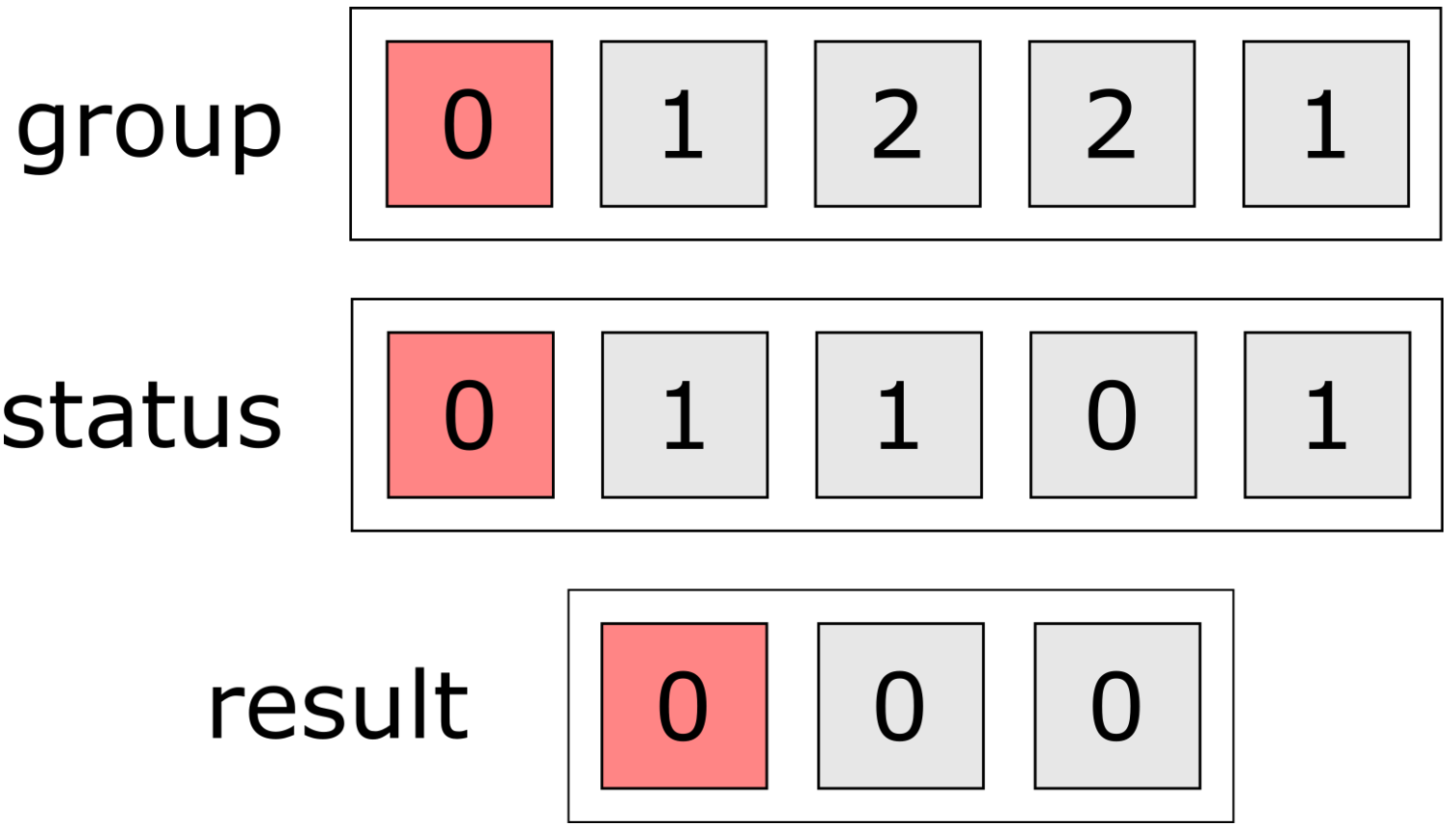
How to benchmark?

Step 2 contd. Enter your own competitor(s)!



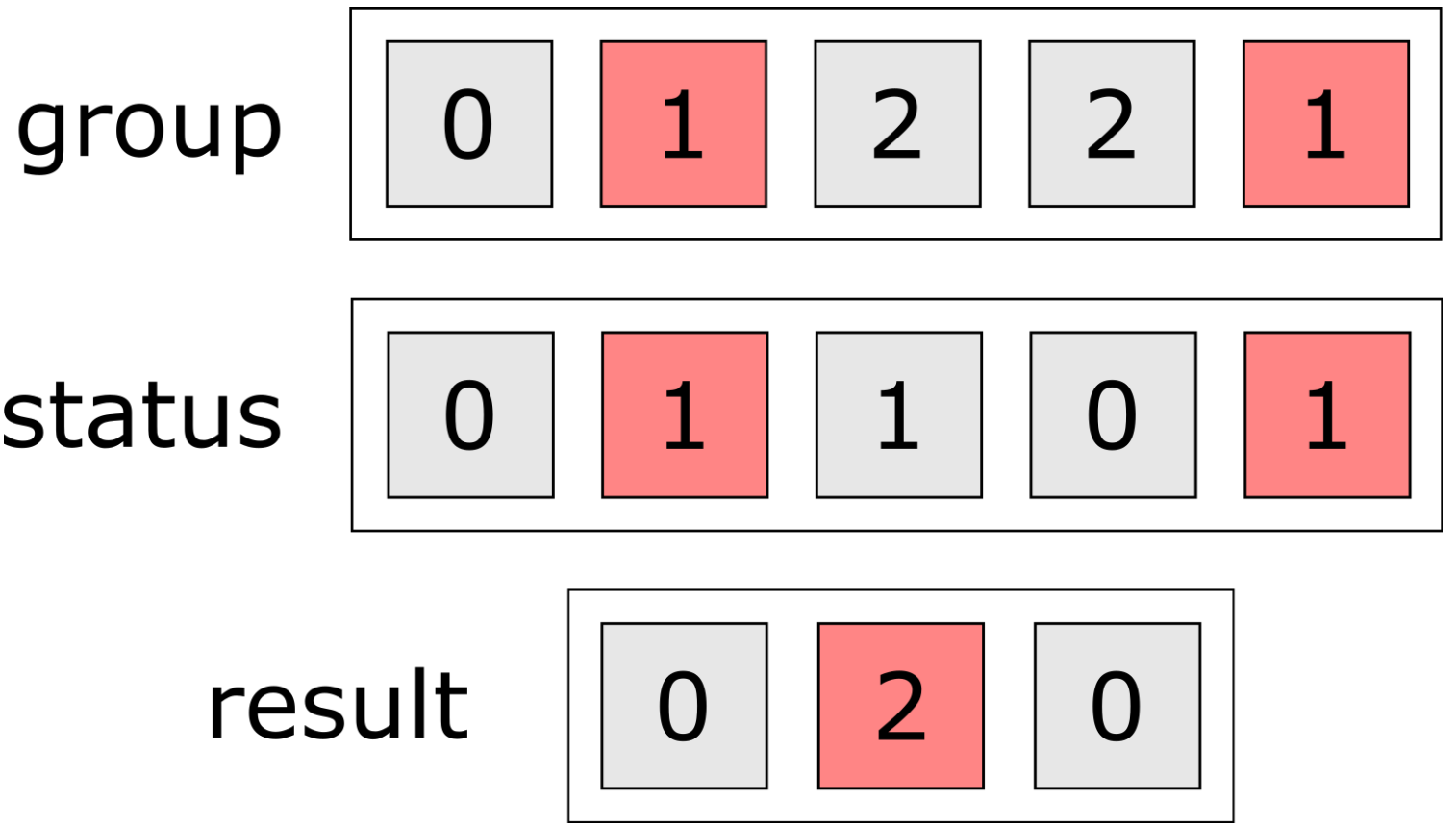
How to benchmark?

Step 2 contd. Enter your own competitor(s)!



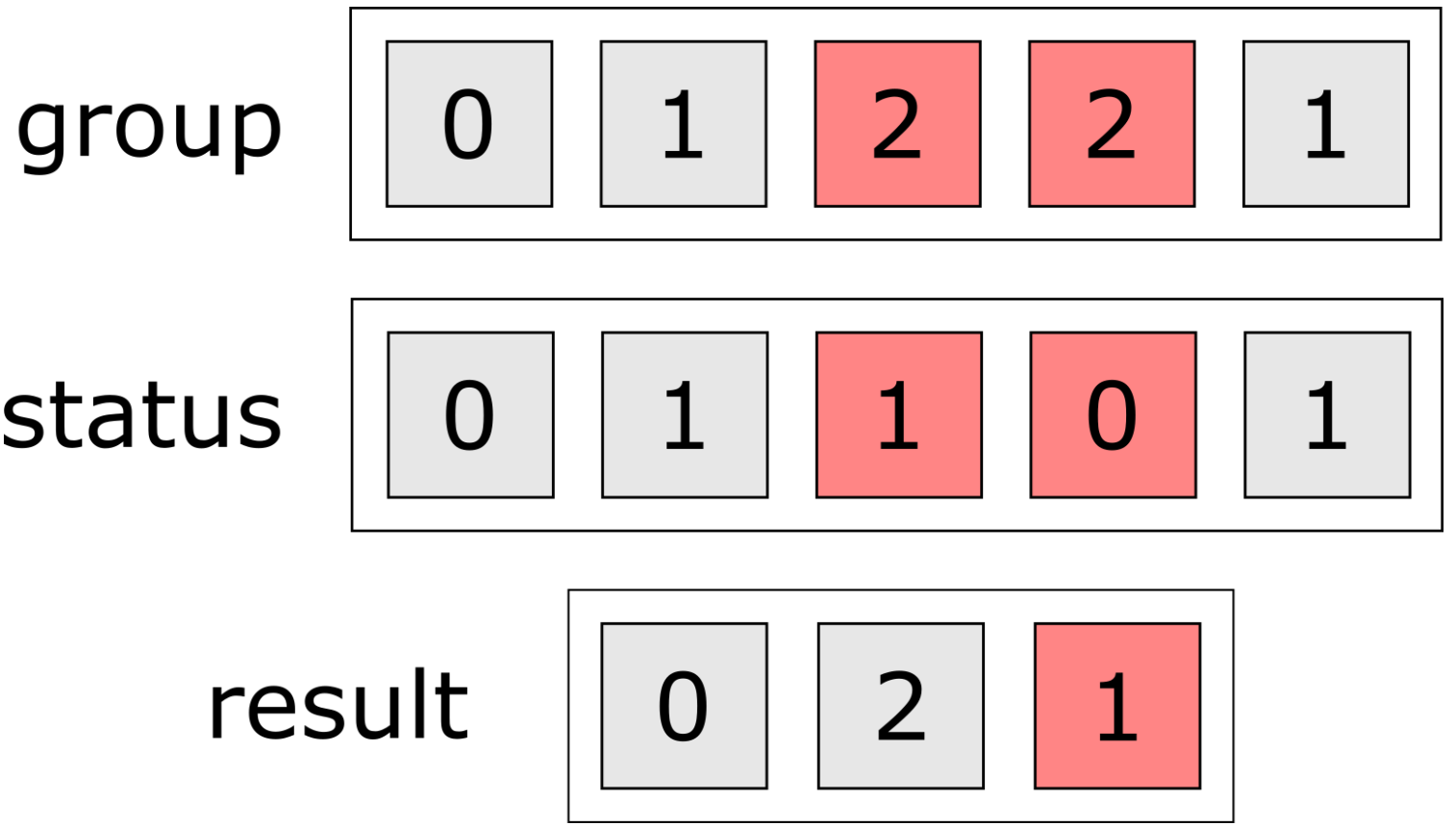
How to benchmark?

Step 2 contd. Enter your own competitor(s)!



How to benchmark?

Step 2 contd. Enter your own competitor(s)!



How to benchmark?

Step 2 contd. Enter your own competitor(s)!

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

// [[Rcpp::export]]
NumericVector rcpp_count_dbl(NumericVector status,
                             NumericVector group,
                             int n_groups) {

 NumericVector out(n_groups);

 for( int i = 0; i < n_groups; i++ ){
  for( int j = 0; j < group.length(); j++ ){
   if(group[j] == i) out[i] += status[j];
  }
 }

 return(out);
}
```

How to benchmark?

Step 3. Off to the races:

```
library(microbenchmark)
microbenchmark(table = table(status, group)[2, ],
               tapply = tapply(status, group, FUN = sum),
               rcpp_count_dbl = rcpp_count_dbl(status, group, 3))
```

Benchmark demonstration: counting events in groups

table(), tapply(), and rcpp_count_dbl()

	Time, milliseconds					
Function	Minimum	25th %	Mean	Median	75th %	Maximum
table	5.5	6.0	7.8	*6.1*	6.3	73
tapply	2.1	2.5	3.2	*2.5*	2.6	81
rcpp_count_dbl	1.6	1.8	2.0	*1.8*	1.8	18

2. Trace your data

What do you mean by trace?

Tracing your data means being notified when

- your data are copied
- your data are cast to a different type

So, why trace?

Copying and casting require additional memory, slowing down your code.

How to trace?

```
tracemem(status)
```

```
## [1] "<000000001A5D2838>"
```

```
tracemem(group)
```

```
## [1] "<00007FF4FD910010>"
```

R yells at me if `status` or `group` are copied or cast to a different type.

Here is a problem

`rcpp_count_dbl` casts both vectors from integer to double

```
rcpp_count_dbl(status, group, n_groups = 3)
```

```
## tracemem[0x00007ff4fd910010 -> 0x00007ff4fd740010]: .Call rcpp_count_dbl e
## tracemem[0x000000001a5d2838 -> 0x00007ff4fd540010]: .Call rcpp_count_dbl e

## [1] 16537 16560 16865
```

Let's fix that

use **integerVector** instead of **numericVector**

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

// [[Rcpp::export]]
IntegerVector rcpp_count_int(IntegerVector status,
                             IntegerVector group,
                             int n_groups) {

 IntegerVector out(n_groups);

 for( int i = 0; i < n_groups; i++ ){
  for( int j = 0; j < group.length(); j++ ){
   if(group[j] == i) out[i] += status[j];
  }
 }

 return(out);
}
```

Better

`tracemem()` has been pacified!

```
rcpp_count_int(status, group, n_groups = 3)
```

```
## [1] 16537 16560 16865
```

Better!

Benchmark demonstration: counting events in groups

table(), tapply(), and Rcpp functions

	Time, milliseconds					
Function	Minimum	25th %	Mean	Median	75th %	Maximum
table	5.5	6.0	7.8	*6.1*	6.3	73
tapply	2.1	2.5	3.2	*2.5*	2.6	81
rcpp_count_dbl	1.6	1.8	2.0	*1.8*	1.8	18
rcpp_count_int	1.4	1.4	1.4	*1.4*	1.4	2.4

3. Count your operations

Why count operations?

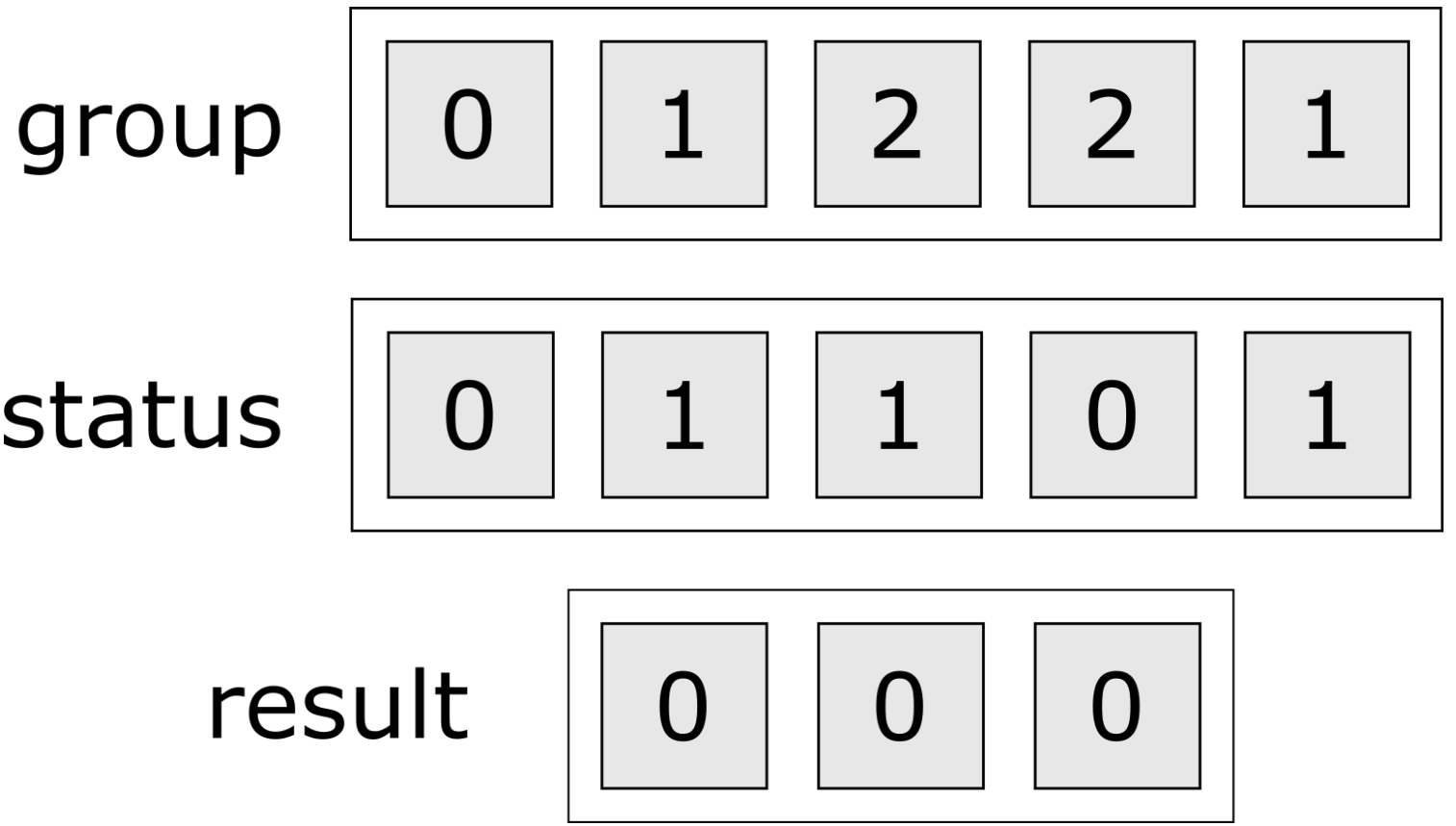
Number of operations $\approx$ computational cost of your code

How to count

You don't have to be exact - think big picture

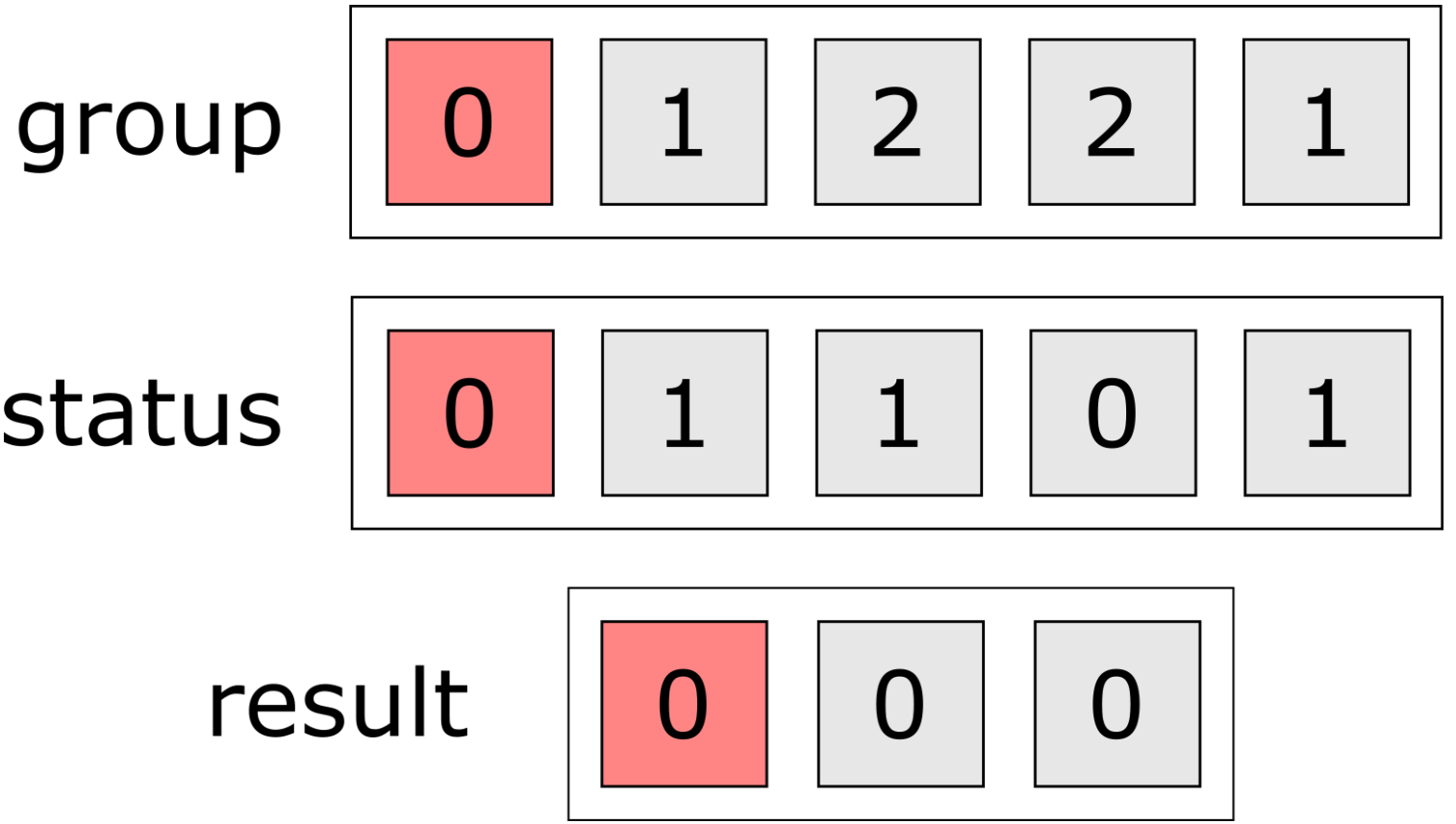
Example

In our C++ function, we use n operations for each unique value in `group`,



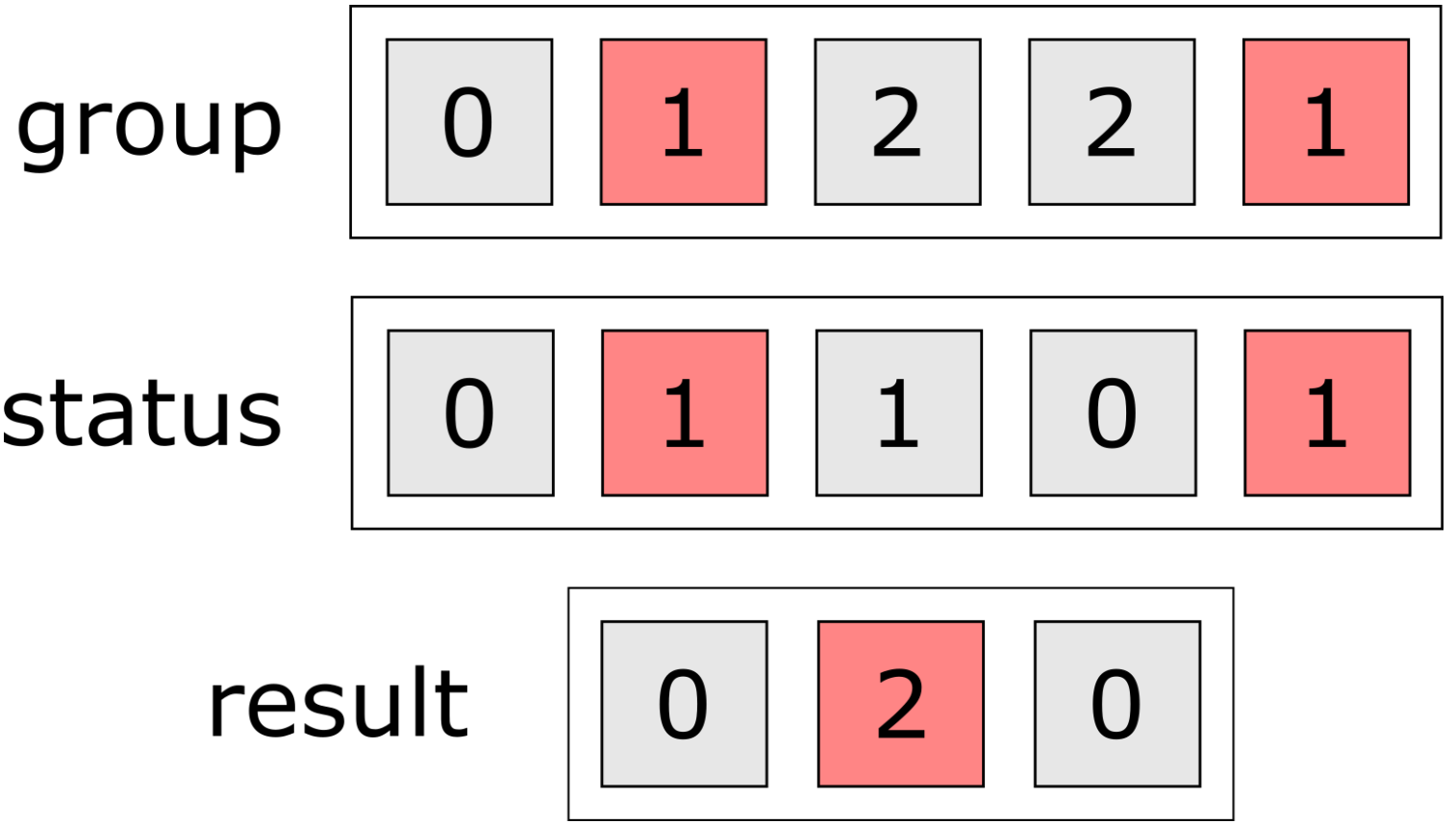
Example

In our C++ function, we use n operations for each unique value in `group`,



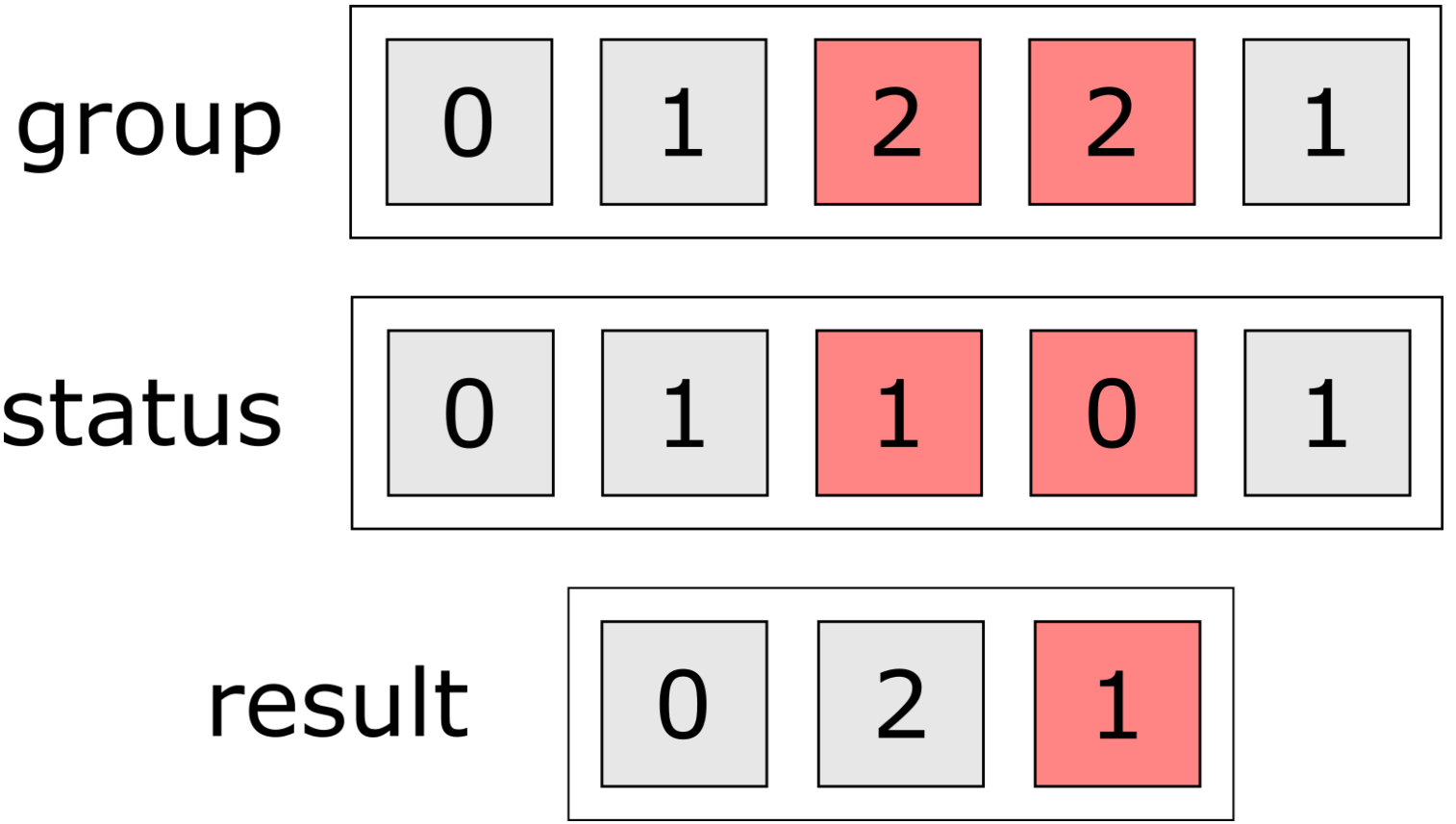
Example

In our C++ function, we use n operations for each unique value in `group`,



Example

In our C++ function, we use n operations for each unique value in `group`,



Example

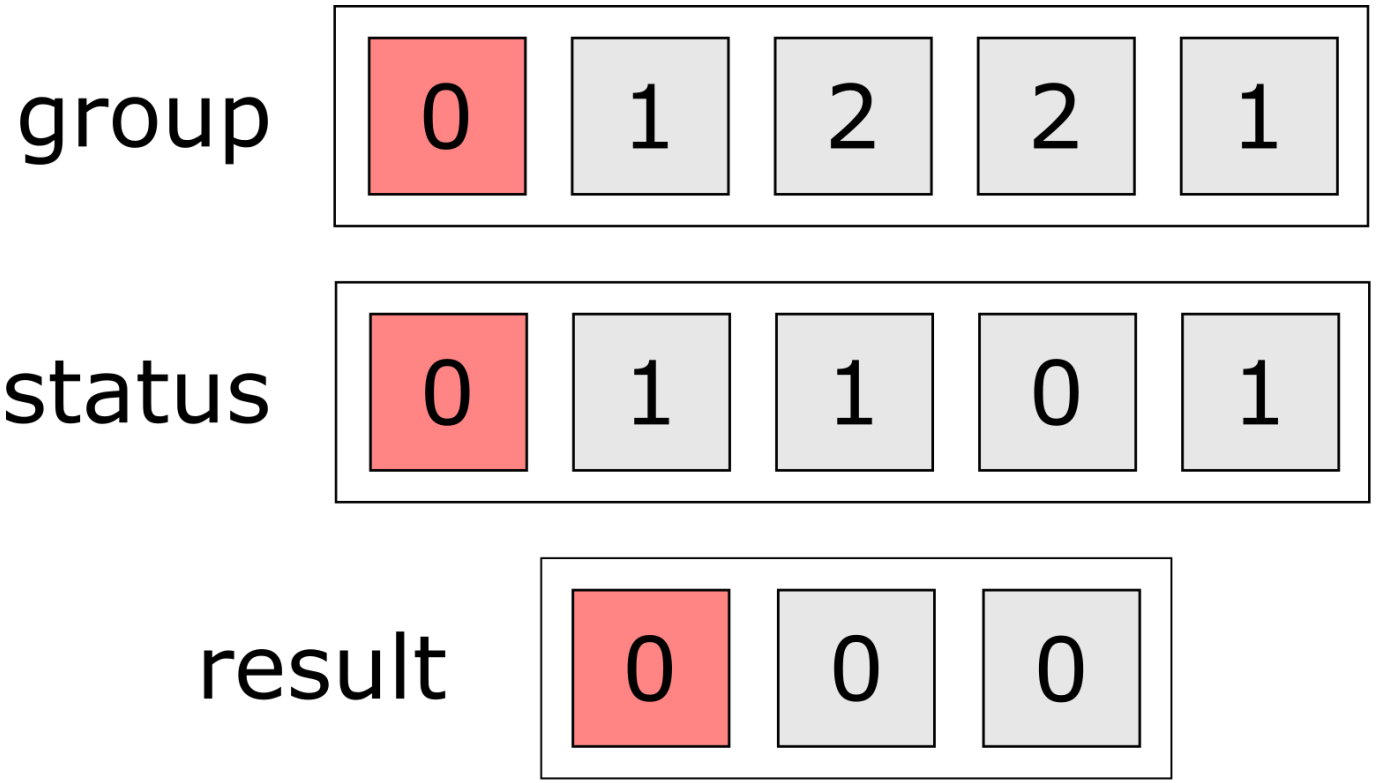
As $n, g \to \infty$, we use $\mathcal{O}(n \cdot g)$ operations, where g = number of groups

Can we reduce the operation cost?

- Can't remove the sum, so we have at least n operations.
- Sum by group without finding group indices?

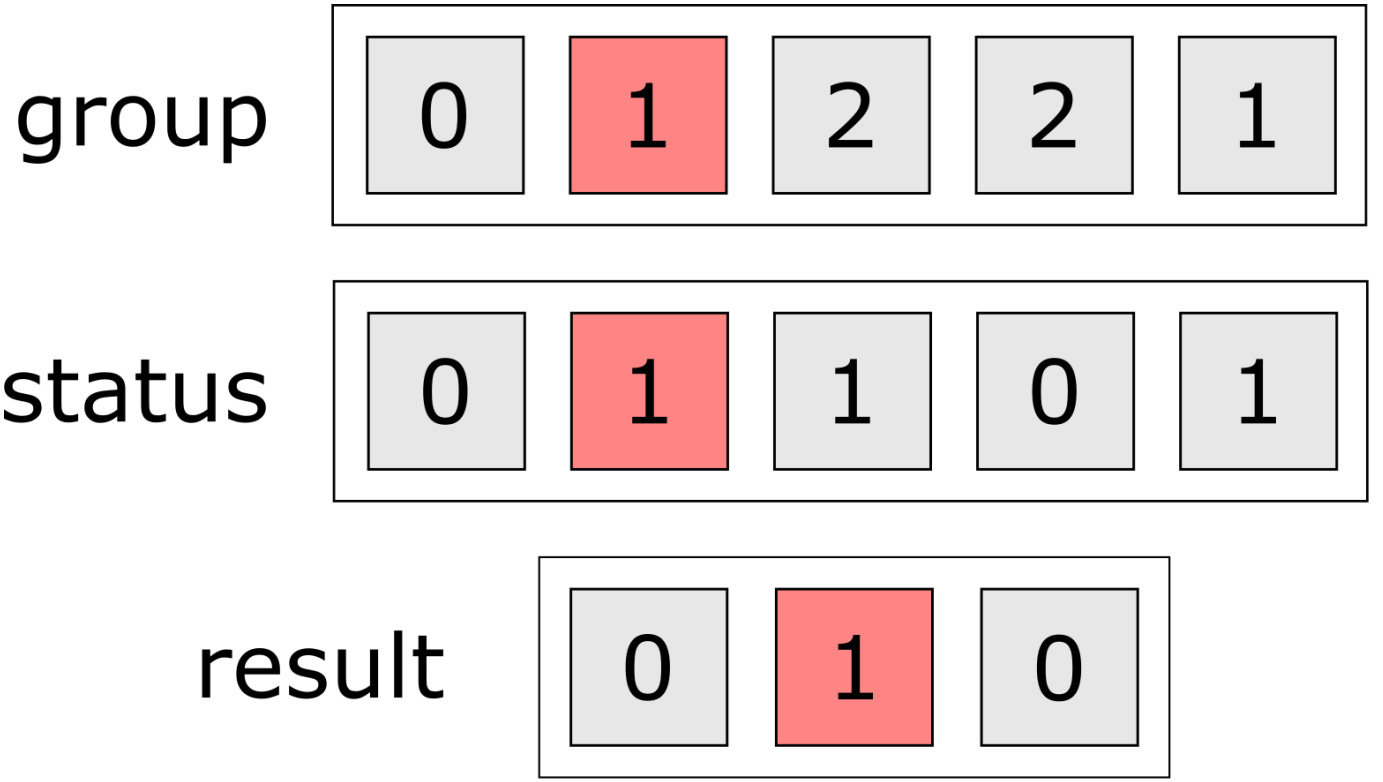
Yes, because the group indices are given to us by the group vector.

1 loop instead of 2



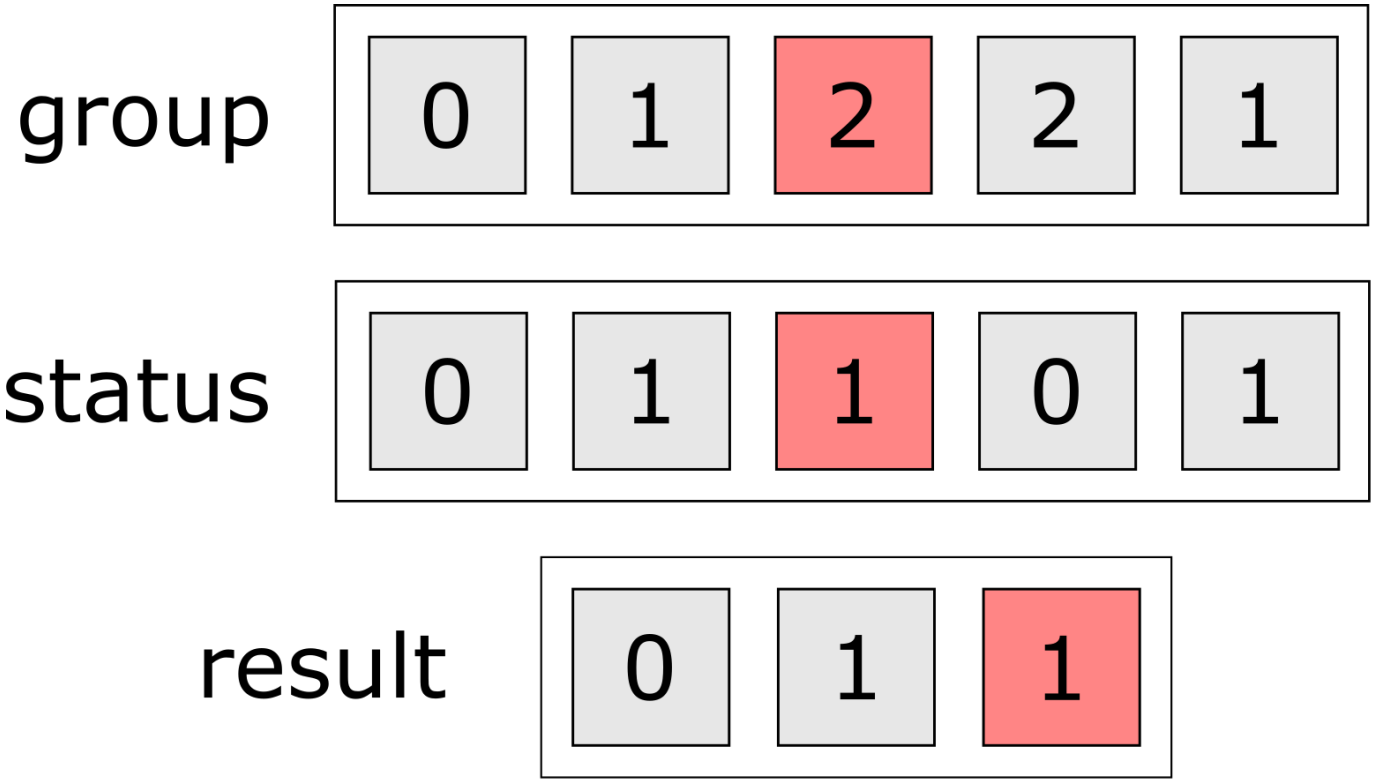

```
result[group[i]] += status[i];
```

1 loop instead of 2



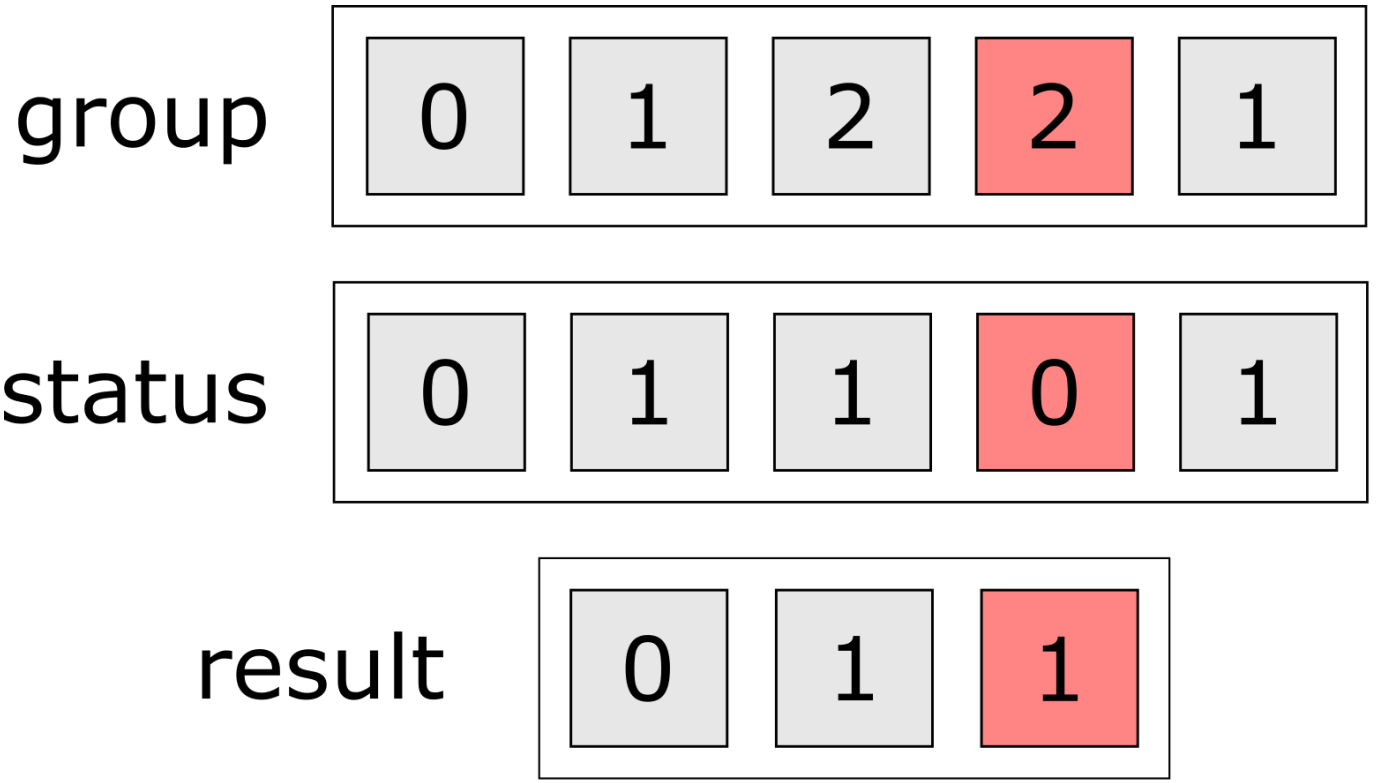

```
result[group[i]] += status[i];
```

1 loop instead of 2



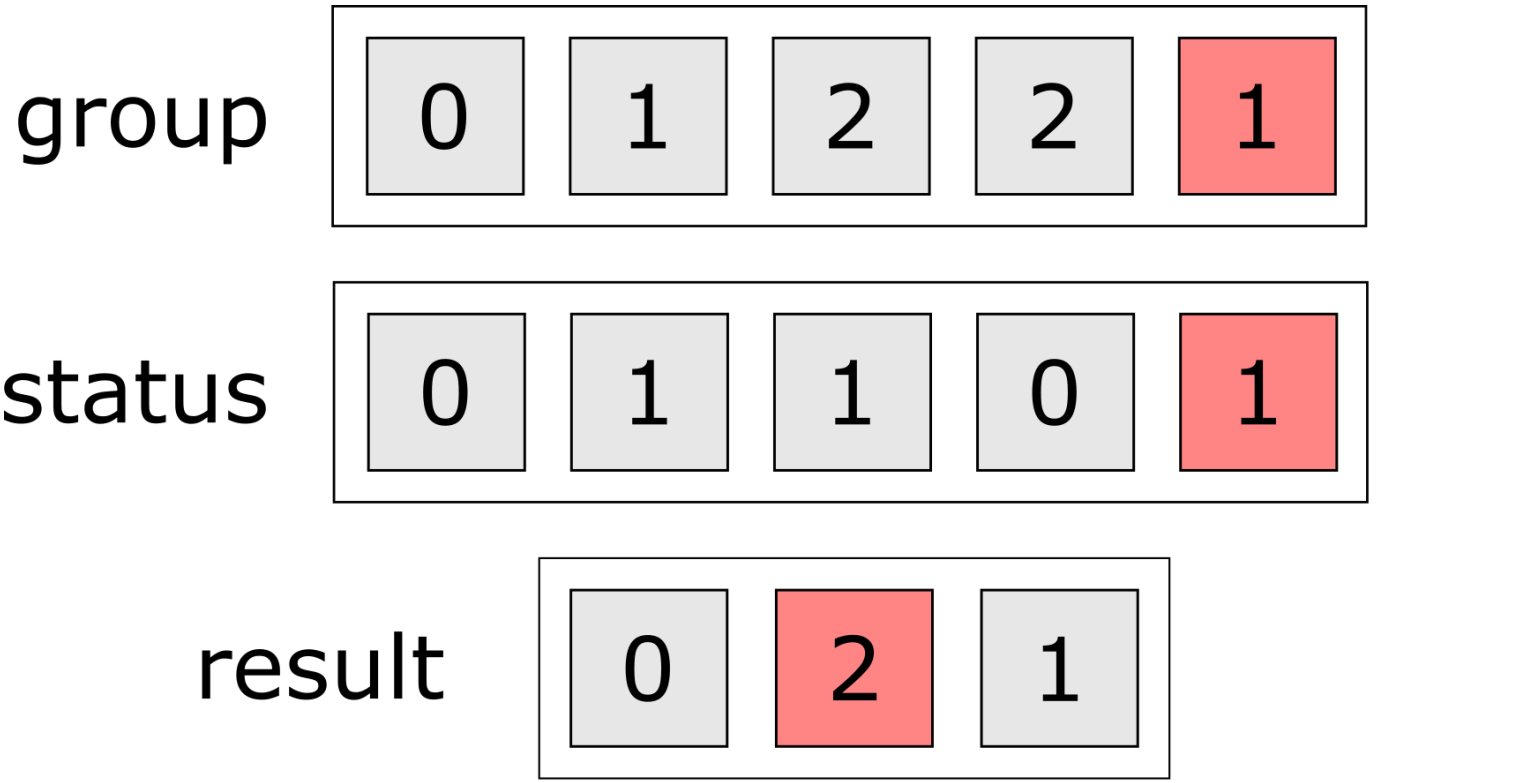

```
result[group[i]] += status[i];
```

1 loop instead of 2




```
result[group[i]] += status[i];
```

1 loop instead of 2




```
result[group[i]] += status[i];
```

1 loop instead of 2

code adapted from `rcpp_count_int`

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

// [[Rcpp::export]]
IntegerVector rcpp_count_1loop_int(IntegerVector status,
                                   IntegerVector group,
                                   int n_groups) {

 IntegerVector out(n_groups);
 IntegerVector::iterator i;
 int j = 0;

 for(i = group.begin() ; i != group.end(); ++i, ++j){
  out[*i] += status[j];
 }

 return(out);
}
```

Much better!

Benchmark demonstration: counting events in groups

table(), tapply(), and Rcpp functions

	Time, milliseconds					
Function	Minimum	25th %	Mean	Median	75th %	Maximum
table	5.5	6.0	7.8	*6.1*	6.3	73
tapply	2.1	2.5	3.2	*2.5*	2.6	81
rcpp_count_dbl	1.6	1.8	2.0	*1.8*	1.8	18
rcpp_count_int	1.4	1.4	1.4	*1.4*	1.4	2.4
rcpp_count_1loop_int	0.18	0.19	0.19	*0.19*	0.19	1.1

4. Ride the Armadillo


aorsf
RcppArmadillo

Armadillo (http://arma.sourceforge.net/)

Armadillo
C++ library for linear algebra & scientific computing

About Documentation Questions Speed Contact Download

- Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use
- Provides high-level syntax and functionality deliberately similar to Matlab
- Useful for algorithm development directly in C++, or quick conversion of research code into production environments
- Provides efficient classes for vectors, matrices and cubes; dense and sparse matrices are supported
- Integer, floating point and complex numbers are supported
- A sophisticated expression evaluator (based on template meta-programming) automatically combines several operations to increase speed and efficiency
- Dynamic evaluation automatically chooses optimal code paths based on detected matrix structures

Armadillo

```
#include <RcppArmadillo.h>
#include <RcppArmadilloExtensions/sample.h>
// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
arma::ivec arma_count_1loop_int(arma::ivec& status,
                                arma::ivec& group,
                                arma::uword n_groups) {

 ivec out(n_groups);
 ivec::iterator i;
 uword j = 0;

 for(i = group.begin() ; i != group.end(); ++i, ++j){
  out[*i] += status[j];
 }

 return(out);

}
```

Better! (almost perfect)

Benchmark demonstration: counting events in groups

table(), tapply(), and Rcpp functions

	Time, milliseconds					
Function	Minimum	25th %	Mean	Median	75th %	Maximum
table	5.5	6.0	7.8	*6.1*	6.3	73
tapply	2.1	2.5	3.2	*2.5*	2.6	81
rcpp_count_dbl	1.6	1.8	2.0	*1.8*	1.8	18
rcpp_count_int	1.4	1.4	1.4	*1.4*	1.4	2.4
rcpp_count_1loop_int	0.18	0.19	0.19	*0.19*	0.19	1.1
arma_count_1loop_int	0.07	0.07	0.08	*0.07*	0.08	1.0
sum	0.05	0.05	0.05	*0.05*	0.05	0.13

Summary

These guidelines were prominently used while I wrote `aorsf`

1. Every function was benchmarked using multiple versions to discover what worked best.

2. `tracemem` was used to discover unintentional copying

3. `glmnet`, the core algorithm of `obliqueRSF`, was replaced with Newton Raphson scoring in `aorsf`. (**Fewer operations & better IPA**).

4. Optimized with `RcppArmadillo`

Thank you!

Incredible team members:

- `aorsf`: Sawyer Welden, Kristin Lenoir, Jaime L. Speiser, Matthew W. Segar, Ambarish Pandey, and Nicholas M. Pajewski
- today's talk: Jaime Lynn Speiser, Joseph Rigdon, Heather Marie Shappell, Nathaniel Sean O'Connell, and Michael Kattan.

Research reported in this presentation was supported[1] by

- Center for Biomedical Informatics, Wake Forest University School of Medicine.
- National Center for Advancing Translational Sciences (NCATS), National Institutes of Health, through Grant Award Number UL1TR001420.

[1] The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.