Despite being on holiday I’m getting in a bit of non-work R coding since the fam has a greater ability to sleep late than I do. Apart from other things I’ve been coming across R package {lutz}, that turns lat/lng pairs into timezone strings.

The package is super neat and has two modes: “fast” (originally based on a {V8}-backed version of and “accurate” using R’s amazing spatial ops.

I ported the javascript algorithm to C++/Rcpp and have been tweaking the bit of package helper code

and extracts the embedded string tree and corresponding timezones array and turns both into something C++ can use. Originally I just made a header file with the same long lines.

but that’s icky and fairly bad form, especially given that C++ will combine adjacent string literals for you.

The stringi::stri\_wrap() function can easily take care of wrapping the time zone array elements for us:

but, I also needed the ability to hard-wrap the encoded string tree at a fixed width. There are lots of ways to do that, here are three of them:

library(Rcpp)

library(stringi)

library(tidyverse)

library(microbenchmark)

library(lutz)

sourceCpp(code = "

#include

// [[Rcpp::export]]

std::vector< std::string > fold\_cpp(const std::string& input, int width) {

int sz = input.length() / width;

std::vector< std::string > out;

out.reserve(sz); // shld make this more efficient

for (unsigned long idx=0; idx

(If you know of a package that has this type of function def leave a note in the comments).

Each one does the same thing: move n sequences of width characters into a new slot in a character vector. Let’s see what they do with this toy long string example:

(src <- paste0(c(rep("a", 30), rep("b", 30), rep("c", 4)), collapse = ""))

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccc"

for (n in c(1, 7, 30, 40)) {

print(fold\_base(src, n))

print(fold\_tidy(src, n))

print(fold\_cpp(src, n))

cat("\n")

}

## [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"

## [18] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "b" "b" "b" "b"

## [35] "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b"

## [52] "b" "b" "b" "b" "b" "b" "b" "b" "b" "c" "c" "c" "c"

## [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"

## [18] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "b" "b" "b" "b"

## [35] "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b"

## [52] "b" "b" "b" "b" "b" "b" "b" "b" "b" "c" "c" "c" "c"

## [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"

## [18] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "a" "b" "b" "b" "b"

## [35] "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b"

## [52] "b" "b" "b" "b" "b" "b" "b" "b" "b" "c" "c" "c" "c"

##

## [1] "aaaaaaa" "aaaaaaa" "aaaaaaa" "aaaaaaa" "aabbbbb" "bbbbbbb"

## [7] "bbbbbbb" "bbbbbbb" "bbbbccc" "c"

## [1] "aaaaaaa" "aaaaaaa" "aaaaaaa" "aaaaaaa" "aabbbbb" "bbbbbbb"

## [7] "bbbbbbb" "bbbbbbb" "bbbbccc" "c"

## [1] "aaaaaaa" "aaaaaaa" "aaaaaaa" "aaaaaaa" "aabbbbb" "bbbbbbb"

## [7] "bbbbbbb" "bbbbbbb" "bbbbccc" "c"

##

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"

## [3] "cccc"

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"

## [3] "cccc"

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"

## [3] "cccc"

##

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbb"

## [2] "bbbbbbbbbbbbbbbbbbbbcccc"

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbb"

## [2] "bbbbbbbbbbbbbbbbbbbbcccc"

## [1] "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbb"

## [2] "bbbbbbbbbbbbbbbbbbbbcccc"

So, we know they all work, which means we can take a look at which one is faster. Let’s compare folding at various widths:

map\_df(c(1, 3, 5, 7, 10, 20, 30, 40, 70), ~{

microbenchmark(

base = fold\_base(src, .x),

tidy = fold\_tidy(src, .x),

cpp = fold\_cpp(src, .x)

) %>%

mutate(width = .x) %>%

as\_tibble()

}) %>%

mutate(

width = factor(width,

levels = sort(unique(width)),

ordered = TRUE)

) -> bench\_df

ggplot(bench\_df, aes(expr, time)) +

ggbeeswarm::geom\_quasirandom(

aes(group = width, fill = width),

groupOnX = TRUE, shape = 21, color = "white", size = 3, stroke = 0.125, alpha = 1/4

) +

scale\_y\_comma(trans = "log10", position = "right") +

coord\_flip() +

guides(

fill = guide\_legend(override.aes = list(alpha = 1))

) +

labs(

x = NULL, y = "Time (nanoseconds)",

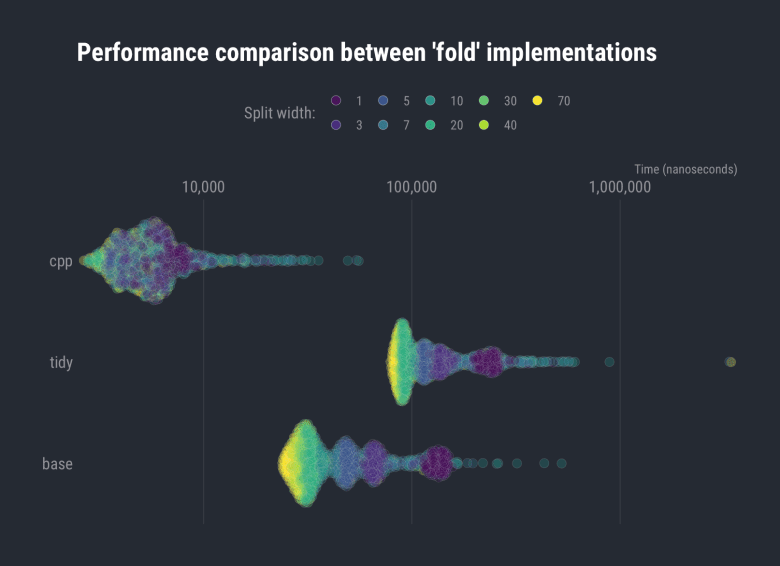
fill = "Split width:",

title = "Performance comparison between 'fold' implementations"

) +

theme\_ft\_rc(grid="X") +

theme(legend.position = "top")

[](https://i1.wp.com/rud.is/b/wp-content/uploads/2019/06/lutz-widths.png?ssl=1)

ggplot(bench\_df, aes(width, time)) +

ggbeeswarm::geom\_quasirandom(

aes(group = expr, fill = expr),

groupOnX = TRUE, shape = 21, color = "white", size = 3, stroke = 0.125, alpha = 1/4

) +

scale\_x\_discrete(

labels = c(1, 3, 5, 7, 10, 20, 30, 40, "Split/fold width: 70")

) +

scale\_y\_comma(trans = "log10", position = "right") +

scale\_fill\_ft() +

coord\_flip() +

guides(

fill = guide\_legend(override.aes = list(alpha = 1))

) +

labs(

x = NULL, y = "Time (nanoseconds)",

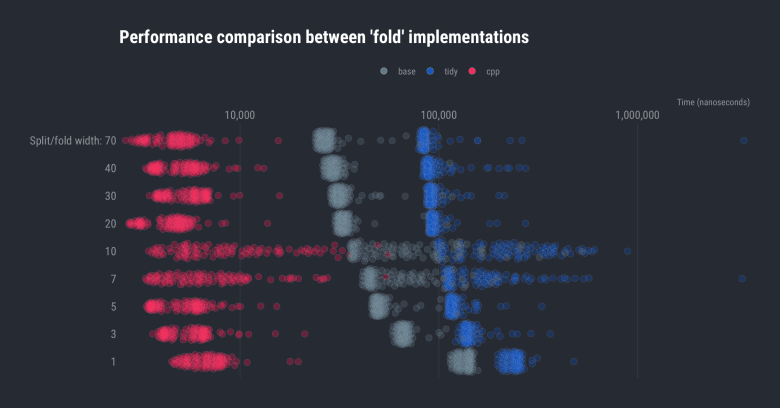
fill = NULL,

title = "Performance comparison between 'fold' implementations"

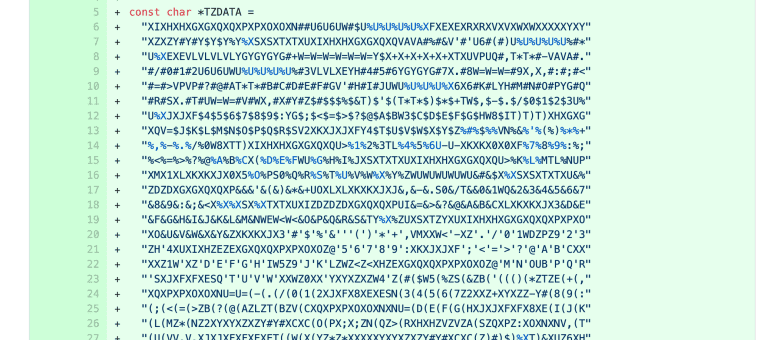
) +

theme\_ft\_rc(grid="X") +

theme(legend.position = "top")

[](https://i2.wp.com/rud.is/b/wp-content/uploads/2019/06/lutz-widths-02.png?ssl=1)

The Rcpp version is both faster and more consistent than the other two implementations (though they get faster as the number of string subsetting operations decrease); but, they’re all pretty fast. For an infrequently run process, it might be better to use the base R version purely for simplicity. Despite that fact, I used the Rcpp version to turn the string tree long line into:

[](https://i0.wp.com/rud.is/b/wp-content/uploads/2019/06/lutz-03.png?ssl=1)

**FIN**

If you have need to “fold” like this how do you currently implement your solution? Found a bug or better way after looking at the code? Drop a note in the comments so you can help others find an optimal solution to their own ‘fold’ing problems.