This post is for all those folk who might be considering wrapping some tidyverse commands in a function to make their code cleaner and hopefully gain some speed advantages.

For this example I’ve created a dummy set of raw data that we can make some arbitrary selection and transformation on and save as clean. In this case I’ve created dates with some observations happening each day, labelled as ‘a’ to ‘c’. Alongside this I’ve duplicated these columns to give us something to drop. Dummy data created with:

library(tidyverse)

# Dummy files to process

dir.create("./temp/raw", recursive=T)

dir.create("./temp/clean")

lapply(1950:2017, function(i){

date = seq.Date(as.Date(paste0(i, "-01-01")),

as.Date(paste0(i, "-12-31")),

by=1)

a = rnorm(length(date))

a1 = rnorm(length(date))

a2 = rnorm(length(date))

b = rpois(length(date), 10)

b1 = rpois(length(date), 10)

b2 = rpois(length(date), 10)

c = rexp(length(date), 5)

c1 = rexp(length(date), 5)

c2 = rexp(length(date), 5)

write\_csv(data.frame(date, a, b, c),

paste0("./temp/raw/file\_", i, ".csv"))

})

We’ve now got a directory with 68 csv files, each containing some fabricated daily data. In order to read files into R, the first thing to do is get a path to it, we can do this with `list.files()`:

# Get a vector of file names

f = list.files("./temp/raw", pattern="file")

Now we have an object, `f`, which contains all our file names we can write a process to get them ready for analysis. I’m illustrating this by selecting 4 columns (date, a, b and c) and converting them to a long tidy format. I’ve had a stab at writing this process in tidyverse alone, but can’t figure out how to pass `write\_csv()` a file name. I suspect the answer lies in turning `f` into a data frame with a column for in and a column for out. Seems pretty awkward to me. I welcome answers in the comments!

# Interactive dplyr

# Who knows what ??? should take, I don't!

system.time(

paste0("./temp/raw/", f) %>%

read\_csv() %>%

select(date, a, b, c) %>%

gather(variable, value, -date) %>%

write\_csv(???)

)

The above doesn’t work, but we can adapt it slightly to make it a function. We’re now able to pass our tidyverse code individual file names, here represented by `i`. Finally, the clean data are written out into the clean folder. In the real world we may also want to change the file name to reflect the clean status.

# As a function

read\_clean\_write = function(i){

paste0("./temp/raw/", i) %>%

read\_csv() %>%

select(date, a, b, c) %>%

gather(variable, value, -date) %>%

write\_csv(paste0("./temp/clean/", i))

}

Finally, we can run the above as a loop (usually bad), lapply or something else:

# Loop

for (j in f){

read\_clean\_write(j)

}

# lapply

lapply(f, read\_clean\_write)

But how fast were they? Can we get faster? Thankfully R provides `system.time()` for timing code execution. In order to get faster, it makes sense to use all the processing power our machines have. The ‘parallel’ library has some great tools to help us run our jobs in parallel and take advantage of multicore processing. My favourite is `mclapply()`, because it is very very easy to take an `lapply` and make it multicore. Note that mclapply doesn’t work on Windows. The following script runs the `read\_clean\_write()` function in a for loop (boo, hiss), lapply and mclapply. I’ve run these as list elements to make life easier later on.

library(parallel)

# Loop

times = list(

loop = system.time(

for (j in f){

read\_clean\_write(j)

}

),

lapply = system.time(

lapply(f, read\_clean\_write)

),

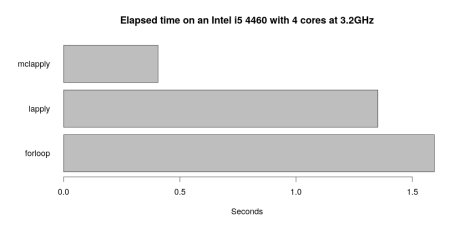
mclapply = system.time(

mclapply(f, mc.cores=4, read\_clean\_write)

)

)

Next we can plot up these results. I’m using sapply to get only the elapsed time from the proc.time object, and then cleaning the elapsed part from the vector name.



Single run comparison of 3 loop methods in R.

x = sapply(times, function(i){i["elapsed"]})

names(x) = substr(names(x), 1, nchar(names(x)) - 8)

par(mar=c(5, 5, 4, 2) + 0.1)

barplot(x, names.arg=names(x),

main="Elapsed time on an Intel i5 4460 with 4 cores at 3.2GHz",

xlab="Seconds", horiz=T, las=1)

par(mar=c(5, 4, 4, 2) + 0.1)

Unsurprisingly mclapply is the clear winner. It’s spreading the work across four cores instead of one, so unless the job is very simple it will always be fastest!

Having run this code a few times, I noticed the results are not consistent. Because we’ve been working in code we can examine the variability. I’ve done this by running each method 100 times:

times = lapply(1:100, function(i){

x = list(

forloop=system.time(

for (j in f){

read\_clean\_write(j)

}

),

lapply = system.time(

lapply(f, read\_clean\_write)

),

mclapply = system.time(

mclapply(f, mc.cores=4, read\_clean\_write)

)

)

x = sapply(x, function(k){k["elapsed"]})

names(x) = substr(names(x), 1, nchar(names(x))-8)

x

})

# Tidy

x = lapply(seq\_along(times), function(i){

data.frame(run=i,

forloop=times[[i]]["forloop"],

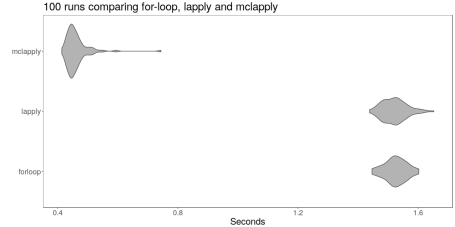
lapply=times[[i]]["lapply"],

mclapply=times[[i]]["mclapply"])

})

x = do.call("rbind.data.frame", x)

My poor computer! Now we can plot these results up. I’ve chosen violin plots to help us see the distribution of results:



Many runs comparing loop methods in R.

png("./temp/violin.png", height=500, width=1000)

x %>%

gather(variable, value, -run) %>%

ggplot(aes(variable, value)) +

geom\_violin(fill="grey70") +

labs(title="100 runs comparing for-loop, lapply and mclapply",

x="",

y="Seconds") +

coord\_flip() +

theme\_bw() +

theme(text = element\_text(size=20),

panel.grid=element\_blank())

dev.off()

Then, we can pull out median values for each:

|  |  |
| --- | --- |
| **Method** | **Time (seconds)** |
| forloop | 1.5255 |
| lapply | 1.5200 |
| mclapply | 0.4515 |

x %>%

gather(variable, value, -run) %>%

group\_by(variable) %>%

summarise(median(value))