R: FuturesStatistics 506

Concurrent and Asynchronous Computing

Asynchronous computing refers to having events that occur independent of the primary control flow in our program.

In a traditional, *synchronous* program each statement or expression *blocks* while evaluating. In other words, it forces the program to wait until if continues. An asynchronous program, in contrast, has some statements that do not block – allowing the program to continue until either (1) the value of the earlier statement is needed or (2) execution resources such as CPU cores are exhausted.

In *parallel* programming we explicitly split portions of our program into chunks of code that can be executed independently. In *concurrent* programming we specify chunks of code that can be executed independently of others. A concurrent program can be executed sequentially or in parallel.

Traditionally concurrent programming has been focused on I/O bound tasks where one is querying external servers or databases and would otherwise have to wait for each query to finish and return before sending the next request. *Concurrency* helps in this situation because it allows the program to wait in multiple queues at once. This video explains how concurrency helps to load webpages more quickly if you're curious.

Concurrent Programming with Futures in R

The R package future provides utilities that allow us to write concurrent programs using an abstraction known as a *future*. Quoting the package author,

> In programming, a *future* is an abstraction for a value that may be available at some point in the future.

Once the future has *resolved*, its *value* becomes available immediately. If we request the value of a future that has not yet resolved the request *blocks* leading our program to wait until the value becomes available.

First, consider the following code:

```
x <- 5
print("Non-trivial code that does not depend on `x`")
```

```
[1] "Non-trivial code that does not depend on `x`"
```

```
x + 4
```

```
[1] 9
```

When we execute those three lines of code, `x` has its value assigned first, then the "non-trivial" code runs, and finally `x` is used in the final statement. However, since `x` is not needed in the "non-trivial" code, we really didn't need to evaluate `x <- 5` until just prior to it's use.

Let's place the `x <- 5` line into concurrent programming. To make things more apparent, we'll add some artificial slow code.

```
library(future)
plan(multisession)
x <- future({
  print("Assigning x")
  Sys.sleep(1)
  5
})
print("Non-trivial code that does not depend on `x`")
```

```
[1] "Non-trivial code that does not depend on `x`"
```

```
value(x) + 4
```

```
[1] "Assigning x"
```

```
[1] 9
```

Take note of the structure of `future()` to assign, and `value()` to access. We'll talk about the `plan` function below.

What's happening here is that when we call `future`, the calculation and assignment of `x` takes place "behind" the other code. Since the calculation of `x` involves that "slow" code, the `print` statement starting with "Non-trivial code" gets executed prior to `x` resolving. Because the final line of code depends on `x` resolving, it is *blocking* and the code will not proceed until `x` is resolved.

Future calls to `value` do not re-resolve the code; instead the result is stored as normal.

```
x <- future({
  print("Assigning x")
  Sys.sleep(1)
  5
})
print("Non-trivial code that does not depend on `x`")
```

```
[1] "Non-trivial code that does not depend on `x`"
```

```
system.time(value(x) + 4)
```

```
[1] "Assigning x"
```

```
   user  system elapsed 
  0.001   0.000   0.988 
```

```
system.time(value(x) - 2)
```

```
[1] "Assigning x"
```

```
   user  system elapsed 
      0       0       0 
```

The first `system.time` is timing not only how long it takes to executer `x + 4`, but also how it takes to finish resolving `x`. Note that the "user" time is 0 in both cases; because of the multi-core approach, the resolving of `x` takes place in another R process, so in *this* session, no processing time was taken (just the waiting time).

`plans`

We used the line `plan(multisession)` above. This tells **futures** how to handle the concurrency. There are three main plans:

1. `plan(sequential)`: This is the default; it basically ignores the future functionality and evaluates them at the point of creation. Useful for debugging.
2. `plan(multicore)`: Behind the scenes this utilizes forking from the **parallel** process.
3. `plan(multisession)`: Behind the scenes this functions similiarly to the socket appraoch, albeit without the need to execute code on each process manually.

Much of the same pros and cons of different approaches applies. Specifically that `plan(multicore)` is not supported on Windows.

An additional complexity is that `plan(multicore)` is not supported in RStudio even in Mac or *nix based systems. The distinction betwee them is less important as **future** handles most of the annoying stuff behind the scenes, but I would still recommend using forking on any system that supports it. You can enable forking on RStudio if you want.

Explicit vs implicit futures

The `future()`/`value()` syntax is a bit burdensome. **future** calls this creating a future "explicitly". We can also create an future "implicitly" with the `%<-%` assignment operator.

```
x %<-% {
  print("Assigning x")
  Sys.sleep(1)
  5
}
print("Non-trivial code that does not depend on `x`")
```

```
[1] "Non-trivial code that does not depend on `x`"
```

```
system.time(x + 4)
```

```
[1] "Assigning x"
```

```
   user  system elapsed 
  0.001   0.000   0.993 
```

```
system.time(x - 2)
```

```
   user  system elapsed 
  0.000   0.000   0.001 
```

In almost all cases, `%<-%` is a drop-in replacement for `<-`. The exception is regarding saving many futures to an object as in a simulation.

```
f <- list()
for (i in 1:3) {
  f[[i]] %<-% i
}
```

```
Error: Subsetting can not be done on a 'list'; only to an environment: 'f[[i]]'
```

```
for (i in 1:3) {
  f[[i]] <- future(i)
}
lapply(f, value)
```

```
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```

It's generally safe to use `%<-%` as default, dropping back to explicit only if needed.

Also, while you don't have to use `{}`, it is strongly recommended that you do. Consider the following:

```
x %<-% 5 * rnorm(1)
```

```
Error in x %<-% 5 * rnorm(1): non-numeric argument to binary operator
```

```r
x %<-% { 5 * rnorm(1) }
```

`%<-%` is a high-priority operator, so `x %<-%` gets evaluated first, before being multiplied.

A similar issue can arise with piping (demonstrated here with R's pipe, but also a problem for the **tidyverse** pipe.)

```r
x %<-% 1:5 |> sum()
```

```
Error in sum(x %<-% 1:5): invalid 'type' (environment) of argument
```

```r
x %<-% { 1:5 |> sum() }
```

Blocking

We defined "blocking" before, but let's reiterate. In the following code:

```r
x %<-% 5   # 1
y <- 3     # 2
x + y      # 3
```

```
[1] 8
```

Because line 3 depends on `x`, it is "blocking". That is, line 2 can be run regardless of whether `x` has finished resolving, but when line 3 is run, it will pause until `x` has resolved.

We can check the status of resolution of a particular future if we desire.

```r
x %<-% {
  Sys.sleep(1)
  5
}
resolved(futureOf(x))
```

```
[1] FALSE
```

```r
Sys.sleep(1.2)
resolved(futureOf(x))
```

```
[1] TRUE
```

The `futureOf` function is necessary when using implicit assignment; we don't need it with explicit assigment:

```
x <- future({
  Sys.sleep(1)
  5
})
resolved(x)
```

```
[1] FALSE
```

```
Sys.sleep(1.2)
resolved(x)
```

```
[1] TRUE
```

Errors in resolution

If a future resolves into an error, the error will get thrown at the point of accessing the object, not during its creation. It will throw the error *every* time the object is accessed.

```
x %<-% {
  stop("error")
}
print("Some code")
```

```
[1] "Some code"
```

```
Sys.sleep(1)
print("more code")
```

```
[1] "more code"
```

```
x + 2
```

```
Error in withCallingHandlers({: error
```

```r
x - 3
```

```
Warning: restarting interrupted promise evaluation
```

```
Error in withCallingHandlers({: error
```

Example 1

Let's use futures to handle data processing. First, let's extract the batting tables from the Lahman database into files by year:

```r
library(DBI)
lahman <- dbConnect(RSQLite::SQLite(), "data/lahman_1871-2022.sqlite")

yrs <- dbGetQuery(lahman,
                  "SELECT DISTINCT yearID FROM BATTING")[, 1]

for (i in yrs) {
  dat <- dbGetQuery(lahman,
                    paste("SELECT * FROM BATTING WHERE yearID ==", i))
  write.csv(dat, file = paste0("data/lahman/batting_", i, ".csv"))
}
```

Make sure it works:

```r
head(dir("data/lahman"))
```

```
[1] "batting_1871.csv" "batting_1872.csv" "batting_1873.csv" "batting_1874.csv"
[5] "batting_1875.csv" "batting_1876.csv"
```

Let's read each file in, collapse it to some averages, then combine the results to run a model upon. Specifically, the question of interest is whether the introduction of the Designated Hitter rule (allowing a player who is not on the field to bat in place of the pitcher) in 1973 shows any correlation with the ratio of RBIs to at bats per season. (Looking at the ratio to account for an increase in the number and length ofa games over time.) To do this, we'll estimate a slope prior to 1973 and a slope post 1973 and compare them.

First, we'll define the function that carries out the data management step.

```r
f <- function(file) {
  dat <- read.csv(file)
  ratios <- dat$RBI/dat$AB
  ratio <- mean(ratios[dat$AB > 0], na.rm = TRUE)
  return(c(dat$yearID[1], ratio))
}
```

Next, run it without futures for timing comparison.

```r
system.time({
  save <- list()
  for (file in dir("data/lahman", full.names = TRUE)) {
    save[[file]] <- f(file)
  }
  savedf <- data.frame(Reduce(rbind, save))
  names(savedf) <- c("year", "ratio")
  savedf$prepost <- savedf$year >= 1973
})
```

```
   user  system elapsed 
  0.278   0.010   0.288 
```

```r
head(save, n = 2)
```

```
$`data/lahman/batting_1871.csv`
[1] 1871.0000000    0.1476379

$`data/lahman/batting_1872.csv`
[1] 1872.0000000    0.1099565
```

```r
head(savedf)
```

```
     year      ratio prepost
init 1871 0.14763790   FALSE
X    1872 0.10995653   FALSE
X.1  1873 0.11487818   FALSE
X.2  1874 0.10178443   FALSE
X.3  1875 0.07594255   FALSE
X.4  1876 0.08076249   FALSE
```

```r
summary(lm(ratio ~ year*prepost, data = savedf))
```

```

Call:
lm(formula = ratio ~ year * prepost, data = savedf)

Residuals:
      Min        1Q    Median        3Q       Max 
-0.024365 -0.007094 -0.000484  0.005685  0.046570 

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)    
(Intercept)       4.566e-01  8.023e-02   5.691 6.56e-08 ***
year             -1.900e-04  4.175e-05  -4.551 1.10e-05 ***
prepostTRUE      -3.227e-01  2.559e-01  -1.261    0.209    
year:prepostTRUE  1.692e-04  1.286e-04   1.315    0.190    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01241 on 148 degrees of freedom
Multiple R-squared:  0.1236,	Adjusted R-squared:  0.1058 
F-statistic: 6.955 on 3 and 148 DF,  p-value: 0.000207
```

```r
system.time({
  plan(multisession)
  save <- list()
  for (file in dir("data/lahman", full.names = TRUE)) {
    save[[file]] <- future(f(file))
  }
  savevals <- lapply(save, value)
  savedf <- data.frame(Reduce(rbind, savevals))
  names(savedf) <- c("year", "ratio")
  savedf$prepost <- savedf$year >= 1973
})
```

```
   user  system elapsed 
  4.207   0.039   6.534 
```

```r
head(save, n = 2)
```

```
$`data/lahman/batting_1871.csv`
MultisessionFuture:
Label: '<none>'
Expression:
f(file)
Lazy evaluation: FALSE
Asynchronous evaluation: TRUE
Local evaluation: TRUE
Environment: R_GlobalEnv
Capture standard output: TRUE
Capture condition classes: 'condition' (excluding 'nothing')
Globals: 2 objects totaling 31.41 KiB (function 'f' of 31.27 KiB, character 'file' of 136 by
Packages: 1 packages ('utils')
L'Ecuyer-CMRG RNG seed: <none> (seed = FALSE)
Resolved: TRUE
Value: 64 bytes of class 'numeric'
Early signaling: FALSE
Owner process: 44cfa7e8-58b8-cd7a-ee10-8503c3a451ad
Class: 'MultisessionFuture', 'ClusterFuture', 'MultiprocessFuture', 'Future', 'environment'

$`data/lahman/batting_1872.csv`
MultisessionFuture:
Label: '<none>'
Expression:
f(file)
Lazy evaluation: FALSE
Asynchronous evaluation: TRUE
Local evaluation: TRUE
Environment: R_GlobalEnv
Capture standard output: TRUE
Capture condition classes: 'condition' (excluding 'nothing')
Globals: 2 objects totaling 31.41 KiB (function 'f' of 31.27 KiB, character 'file' of 136 by
Packages: 1 packages ('utils')
L'Ecuyer-CMRG RNG seed: <none> (seed = FALSE)
Resolved: TRUE
Value: 64 bytes of class 'numeric'
Early signaling: FALSE
Owner process: 44cfa7e8-58b8-cd7a-ee10-8503c3a451ad
Class: 'MultisessionFuture', 'ClusterFuture', 'MultiprocessFuture', 'Future', 'environment'
```

```r
head(savevals, n = 2)
```

```
$`data/lahman/batting_1871.csv`
[1] 1871.0000000    0.1476379

$`data/lahman/batting_1872.csv`
[1] 1872.0000000    0.1099565
```

```r
head(savedf)
```

```
     year      ratio prepost
init 1871 0.14763790   FALSE
X    1872 0.10995653   FALSE
X.1  1873 0.11487818   FALSE
X.2  1874 0.10178443   FALSE
X.3  1875 0.07594255   FALSE
X.4  1876 0.08076249   FALSE
```

```r
summary(lm(ratio ~ year*prepost, data = savedf))
```

```

Call:
lm(formula = ratio ~ year * prepost, data = savedf)

Residuals:
      Min        1Q    Median        3Q       Max 
-0.024365 -0.007094 -0.000484  0.005685  0.046570 

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)    
(Intercept)       4.566e-01  8.023e-02   5.691 6.56e-08 ***
year             -1.900e-04  4.175e-05  -4.551 1.10e-05 ***
prepostTRUE      -3.227e-01  2.559e-01  -1.261    0.209    
year:prepostTRUE  1.692e-04  1.286e-04   1.315    0.190    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01241 on 148 degrees of freedom
Multiple R-squared:  0.1236,	Adjusted R-squared:  0.1058 
F-statistic: 6.955 on 3 and 148 DF,  p-value: 0.000207
```

This is a case where parallel processing hurts us - because the operation is so fast. What if we had slower code?

Artificially slower code

Let's add an artificial bottleneck by pausing 1 second within each file. This mimics the data-processing being on larger data or just overall slower.

```
f <- function(file) {
  Sys.sleep(1)
  dat <- read.csv(file)
  ratios <- dat$RBI/dat$AB
  ratio <- mean(ratios[dat$AB > 0], na.rm = TRUE)
  return(c(dat$yearID[1], ratio))
}
```

```
system.time({
  save <- list()
  for (file in dir("data/lahman", full.names = TRUE)) {
    save[[file]] <- f(file)
  }
  savedf <- data.frame(Reduce(rbind, save))
  names(savedf) <- c("year", "ratio")
  savedf$prepost <- savedf$year >= 1973
})
```

```
   user  system elapsed 
  1.615   0.085 154.298 
```

```
system.time({
  plan(multisession)
  save <- list()
  for (file in dir("data/lahman", full.names = TRUE)) {
    save[[file]] <- future(f(file))
  }
  savevals <- lapply(save, value)
  savedf <- data.frame(Reduce(rbind, savevals))
  names(savedf) <- c("year", "ratio")
  savedf$prepost <- savedf$year >= 1973
})
```

```
   user  system elapsed 
  5.842   0.078  21.165 
```

Example 2

Here's another example of a simulation. Let's compare what happens in a large logistic regression model when there are many informative variables, as opposed to when there is only one variable related to the outcome and the rest add independent noise. Specifically, how does the distribution of the coefficient on the variable of interest change between the two situations

```
n <- 1000
p <- 100

x <- matrix(rnorm(n * p), ncol = p)

pr1 <- arm::invlogit(rowSums(x)) # all x's informative
pr2 <- arm::invlogit(x[,1]) # Only x1 informative

f <- function(x, pr) {
  y <- rbinom(nrow(x), 1, prob = pr)
  suppressWarnings(mod <- glm(y ~ x, family=binomial()))
  return(mod$coef[2])
}

reps <- 100

save1 <- list()
save2 <- list()
system.time(for (i in seq_len(reps)) {
  save1[[i]] <- future(f(x, pr1), seed = TRUE)
  save2[[i]] <- future(f(x, pr2), seed = TRUE)
})
```

```
   user  system elapsed
  5.973   0.098   8.639
```

```
save1 <- sapply(save1, value)
save2 <- sapply(save2, value)

# Use median and IQR because `save1` likely has some extreme values
matrix(c(median(save1), median(save2), IQR(save1), IQR(save2)),
       byrow = TRUE, nrow = 2,
       dimnames = list(c("median", "IQR"), c("all x", "only x1")))
```

```
          all x   only x1
median 68.09187 1.1249493
IQR    81.84375 0.1228094
```

Note the use of `seed = TRUE` to `future`. This is needed if RNG is used within a future to avoid issues with non-random results.

For comparison, timing of a non-futures version.

```
save1 <- list()
save2 <- list()
system.time(for (i in seq_len(reps)) {
  save1[[i]] <- f(x, pr1)
  save2[[i]] <- f(x, pr2)
})
```

```
   user  system elapsed
 19.950   0.168  20.209
```