R: Parallel ProcessingStatistics 506

Serial versus Parallel processing

Consider that we have a series of functions to run:, `f1`, `f2`, `f3`, etc.

Serial processing means that `f1` runs first, and until `f1` completes, nothing else can run. Once `f1` completes, `f2` begins, and the process repeats.

Parallel processing (in the extreme) means that all the `f#` functions start simultaneously and run to completion on their own.

In the past, in order to run each of these functions at the same time, we'd require multiple computers. Most (all?) modern are multicore, meaning that we split the processing across these cores.

The serial-parallel scale

A problem can range from "inherently serial" to "perfectly parallel".

An "inherently serial" problem is one which cannot be parallelized at all - for example, if `f2` depended on the output of `f1` before it could begin, using parallelization would not help. In fact, it may actually run slower than on a single core, due to the pre- and post-processing required for paralell performance.

On the other hand a "perfectly parallel" problem (often called embarrassingly parallel) is one in which there is absolutely no dependency between iterations, such that all functions can start simultaneously. Monte carlo simulations usually (though not always) fall into this category, as well as the `*apply` functions.

Sometimes the dependency occurs at the end of each function; so we could start running `f1` and `f2` in completion, but `f2` would pause before finishing while it waits for `f1` to finish. We can still see a speed gain here.

Here we're going to exclusively discuss the perfectly parallel situation; it is where most (though of course not all) statistical situations will land. The more complicated parallel situations arise generally in deeper computer science scenarios, for example handling many users of a service.

Terminology

Let's just nail down some terminology.

- A *core* is a general term for either a single processor on your own computer (technically you only have one processor, but a modern processor like the i7 can have multiple cores - hence the term) or a single machine in a cluster network.
- A *cluster* is a collection of objecting capable of hosting cores, either a network or just the collection of cores on your personal computer.
- A *process* is a single running version of R (or more generally any program). Each core runs a single process. Typically a *process* runs a single function.

The `parallel` package

There are a number of packages which can be used for parallel processing in R. Two of the earliest and strongest were `multicore` and `snow`. However, both were adopted in the base R installation and merged into the `parallel` package.

```
library(parallel)
```

You can easily check the number of cores you have access to with `detectCores`:

```
detectCores()
```

```
[1] 8
```

The number of cores represented is not neccessarily correlated with the number of processors you actually have thanks to the concept of "logical CPUs". For the most part, you can use this number as accurate. Trying to use more cores than you have available won't provide any benefit.

Methods of Paralleization

There are two main ways in which code can be parallelized, via *sockets* or via *forking*. These function slightly differently:

- The *socket* approach launches a new version of R on each core. Technically this connection is done via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer. I mention this because you may get a warning from your computer asking whether to allow R to accept incoming connections, you should allow it.

- The *forking* approach copies the entire current working instance of R and moves it to a new core.

There are various pro's and con's to the two approaches:

Socket:

- Pro: Works on any system (including Windows).
- Pro: Each process on each node is unique so it can't cross-contaminate.
- Con: Each process is unique so it will be slower since you can't do any general pre-processing prior to splitting into parallel processing.
- Con: Because of this, things such as package loading need to be done in each process separately. Variables defined on your main version of R don't exist on each core unless explicitly placed there.
- Con: More complicated to implement.

Forking:

- Con: Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
- Con: Because processes are duplicates, it can cause issues specifically with random number generation (which should usually be handled by `parallel` in the background) or when running in a GUI (such as RStudio). This doesn't come up often, but if you get odd behavior, this may be the case.
- Pro: Faster than sockets.
- Pro: Because it copies the existing version of R, your entire workspace exists in each process.
- Pro: Trivially easy to implement.

In general, I'd recommend using forking if you're not on Windows.

Note: These notes were compiled on OS X.

Parallel processing in RStudio

A mentioned "con" of forking about is that GUI applications (such as RStudio) can behave oddly with forking. This does not mean that forking won't work in RStudio, but it does mean that you may get crashing or other undesired results. You can use the actual R GUI or the command line if you want to avoid this issue.

Note that the **parallel** package will still let you carry out forking in RStudio (and really, it works most of the time). There are other packages, such as **future** that we'll talk about next, that explicitly disable forking in RStudio. The **parallelly** package can detect whether your current R session supports forking:

```r
library(parallelly)
supportsMulticore()
```

```
[1] TRUE
```

Try running that on RStudio. These notes are not build inside RStudio so in this case, forking is supported.

Forking with `mclapply`

The most straightforward way to enable parallel processing is by switching from using `lapply` to `mclapply`.

```r
library(lme4)
```

```
Loading required package: Matrix
```

```r
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time(lapply(1:1000, f))
```

```
   user  system elapsed 
  5.609   0.032   5.683 
```

```r
system.time(mclapply(1:1000, f))
```

```
   user  system elapsed 
  3.599   0.236   3.870 
```

If you were to run this code on Windows, `mclapply` would simply call `lapply`, so the code works but sees no speed gain.

`mclapply` takes an argument, `mc.cores`. By default, `mclapply` will use all cores available to it. If you don't want to (either becaues you're on a shared system or you just want to save processing power for other purposes) you can set this to a value lower than the number of cores you have. Setting it to 1 disables parallel processing, and setting it higher than the number of available cores has no effect.

```r
system.time(mclapply(1:100, f, mc.cores = 1))
```

```
  user  system elapsed 
 0.529   0.007   0.536 
```

```r
system.time(mclapply(1:100, f, mc.cores = 2))
```

```
  user  system elapsed 
 0.347   0.092   0.440 
```

```r
system.time(mclapply(1:100, f, mc.cores = 4))
```

```
  user  system elapsed 
 0.217   0.101   0.324 
```

```r
system.time(mclapply(1:100, f, mc.cores = 8))
```

```
  user  system elapsed 
 0.698   0.278   0.302 
```

```r
system.time(mclapply(1:100, f, mc.cores = 16))
```

```
  user  system elapsed 
 0.679   0.395   0.390 
```

Operating on lists

There are two functions that are very handy for operating on lists, `Reduce` and `do.call`. These come in useful because there is no analogue of `sapply` for automatically collapsing lists.

`Reduce` takes in a function and a list, and applies the function to elements of the list, recursively.

```r
l <- list(1, 2, 3, 4, 5)
Reduce(sum, l)
```

```
[1] 15
```

```r
Reduce(`+`, l)
```

```
[1] 15
```

Those calls are equivalent to

```r
sum(1, sum(2, sum(3, sum(4, 5))))
```

```
[1] 15
```

```r
(1 + (2 + (3 + (4 + 5))))
```

```
[1] 15
```

```r
`+`(1, `+`(2, `+`(3, `+`(4, 5))))
```

```
[1] 15
```

On the other hand, `do.call` passes all elements of the list in as arguments to the function:

```r
do.call(sum, l)
```

```
[1] 15
```

```r
do.call(`+`, l)
```

```
Error in .Primitive("+")(1, 2, 3, 4, 5): operator needs one or two arguments
```

The equivalent calls are

```r
sum(1, 2, 3, 4, 5)
```

```
[1] 15
```

```r
`+`(1, 2, 3, 4, 5)
```

```
Error in `+`(1, 2, 3, 4, 5): operator needs one or two arguments
```

The `+` function takes in exactly two arguments, so it doens't work with `do.call` (unless the list has two elements only). `sum` takes in any number of arguments (via `...`) so it works in either case.

A common use case is to reduce a list of vectors down to a matrix or a larger vector.

```r
head(mt <- as.list(mtcars[, -1]), 2)
```

```
$cyl
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4

$disp
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
[13] 275.8 275.8 472.0 460.0 440.0  78.7  75.7  71.1 120.1 318.0 304.0 350.0
[25] 400.0  79.0 120.3  95.1 351.0 145.0 301.0 121.0
```

```r
head(do.call(cbind, mt))
```

```
     cyl disp  hp drat    wt  qsec vs am gear carb
[1,]   6  160 110 3.90 2.620 16.46  0  1    4    4
[2,]   6  160 110 3.90 2.875 17.02  0  1    4    4
[3,]   4  108  93 3.85 2.320 18.61  1  1    4    1
[4,]   6  258 110 3.08 3.215 19.44  1  0    3    1
[5,]   8  360 175 3.15 3.440 17.02  0  0    3    2
[6,]   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
head(Reduce(cbind, mt))
```

```
     init
[1,]    6 160 110 3.90 2.620 16.46 0 1 4 4
[2,]    6 160 110 3.90 2.875 17.02 0 1 4 4
[3,]    4 108  93 3.85 2.320 18.61 1 1 4 1
[4,]    6 258 110 3.08 3.215 19.44 1 0 3 1
[5,]    8 360 175 3.15 3.440 17.02 0 0 3 2
[6,]    6 225 105 2.76 3.460 20.22 1 0 3 1
```

Note the differences in how they operate with respect to column names. Because of the recursive nature of `Reduce`, the names get lost.

```r
head(do.call(c, mt))
```

```
cyl1 cyl2 cyl3 cyl4 cyl5 cyl6
   6    6    4    6    8    6
```

```r
head(Reduce(c, mt))
```

```
[1] 6 6 4 6 8 6
```

The reverse issue arises - because `do.call` takes in all elements simultaneously, it keeps the names around.

There's also a huge performance difference between these functions, but it's not always clear (at least to me) which one would be faster. We'd expect `do.call` to always be faster since it only calls its function once (whereas `Reduce` calls it `length(list) - 1` times), but in some cases, it's far slower.

```r
library(microbenchmark)
microbenchmark(
  do.call(cbind, mt),
  Reduce(cbind, mt),
  do.call(c, mt),
  Reduce(c, mt)
)
```

```
Warning in microbenchmark(do.call(cbind, mt), Reduce(cbind, mt), do.call(c, :
less accurate nanosecond times to avoid potential integer overflows
```

```
Unit: microseconds
               expr    min      lq     mean median      uq    max neval
 do.call(cbind, mt)  1.968  2.1320  2.30994  2.255  2.3985  4.428   100
  Reduce(cbind, mt)  7.995  8.5075  8.97244  8.774  9.1840 17.056   100
     do.call(c, mt) 25.748 26.1990 26.94315 26.445 26.8345 44.280   100
      Reduce(c, mt)  4.920  5.2275  5.53828  5.494  5.7195 10.291   100
```

Example

Let's see an example. Bootstrapping is a technique that can be used to estimate standard errors for any statistic. It is a computationally expensive operation that is useful in situations where you don't want to make a distributional assumption for your statistic (e.g. you think the data is very non-normal and don't want to use the central limit theorem).

The basic idea is to resample your data, with replacement, treating your original sample as a "population". You can then estimate your statistic on each bootstrap sample, and estimate the standard error from the empirial distribution of bootstrap estimates of the statistic.

Let θ be the parameter of interest from some population, and let $\hat{\theta}$ be the estimate of θ from data X. We can obtain a bootstrap estimate of θ, called $\tilde{\theta}$, by estimating θ in the data $\tilde{X}$, where $\tilde{X}$ is a resampling with replacement of X. Noteably, the dimension of X and $\tilde{X}$ are identical.

Repeat this process R times, leading to $\tilde{\theta}_r$ for $r \in [1, R]$. (Typically $R > 1000$.) From this, we can estimate the standard error of $\hat{\theta}$ as

$$\sqrt{\frac{1}{R-1}\sum_{i=1}^{R}(\tilde{\theta}_r - \overline{\theta}_R)^2},$$

where $\overline{\theta}_R = \frac{\sum_{i=1}^{R}\tilde{\theta}_r}{R}$. Note that the best estimate of θ is $\hat{\theta}$, **not** $\overline{\theta}_R$.

There are no distributional assumptions in this derivation.

Let's load up the 2020 RECS data.

```r
recs <- read.csv("data/recs2020_public_v6.csv")
names(recs) <- tolower(names(recs))
```

We're going to fit a model predicting the electricty expenses of buildings, controlling for the "heating degree days" and "cooling degree days" and their interaction as well as the number of rooms. Our goal is to examine whether costs differ between mobile homes and single family homes. Since the data is national, we're concerned about differences by states (either due to climate or costs), so we'll include state as a random effect, along with "heating degree days" and "cooling degree days" (and their interaction) random slopes.

First, standardize the continuous variables (they're on very different scales which can cause converge issues):

```r
recs$hdd65 <- with(recs, (hdd65 - mean(hdd65))/sd(hdd65))
recs$cdd65 <- with(recs, (cdd65 - mean(cdd65))/sd(cdd65))
recs$dollarel <- with(recs, (dollarel - mean(dollarel))/sd(dollarel))
```

```
recs$typehuq <- as.factor(recs$typehuq)
```

Fit the model on the original data.

```
library(lme4)
mod <- lmer(dollarel ~ hdd65*cdd65 + typehuq + totrooms +
              (1 + hdd65*cdd65 | state_fips), data = recs)
```

```
Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl = control$checkConv, :
Model failed to converge with max|grad| = 0.00387431 (tol = 0.002, component 1)
```

```
# typehuq 1 (reference) is mobile home, typehuq2 2 is single family
estcoef <- fixef(mod)["typehuq2"]
```

Now we define the bootstrap function:

```
boot <- function(dat) {
  dat <- dat[sample(1:nrow(dat), replace = TRUE), ]
  mod <- lmer(dollarel ~ hdd65*cdd65 + typehuq + totrooms +
                (1 + hdd65*cdd65 | state_fips), data = dat)
  return(fixef(mod)["typehuq2"])
}
```

Finally, we use `lapply` and `mclapply`. For demonstration purposes here, we'll keep the number of reps low; in practice you want a much higher number of reps (at least 1,000).

```
reps <- 10
system.time(res1 <- lapply(seq_len(reps), function(x) boot(recs)))
```

```
   user  system elapsed
 16.311   1.023  17.354
```

```
system.time(res2 <- mclapply(seq_len(reps), function(x) boot(recs)))
```

```
   user  system elapsed
  8.333   0.130   9.702
```

```r
sd1 <- sd(Reduce(c, res2))
sd2 <- sd(Reduce(c, res2))
rbind(c(estcoef + 1.96*sd1, estcoef - 1.96*sd1),
      c(estcoef + 1.96*sd2, estcoef - 1.96*sd2))
```

```
        typehuq2   typehuq2
[1,] -0.04709568 -0.1778196
[2,] -0.04709568 -0.1778196
```

Using sockets with `parLapply`

As promised, the sockets approach to parallel processing is more complicated and a bit slower, but works on Windows systems. The general process we'll follow is

1. Start a cluster with as many nodes as desired.
2. Execute any pre-processing code necessary in each node (e.g. loading a package)
3. Use `par*apply` as a replacement for `*apply`. Note that unlike `mcapply`, this is *not* a drop-in replacement.
4. Destroy the cluster (not necessary, but best practices).

Starting a cluster

The function to start a cluster is `makeCluster` which takes in as an argument the number of cores:

```r
numCores <- detectCores()
numCores
```

```
[1] 8
```

```r
cl <- makeCluster(numCores)
```

The function takes an argument `type` which can be either `PSOCK` (the socket version) or `FORK` (the fork version). Generally, `mclapply` should be used for the forking approach, so there's no need to change this.

If you were running this on a network of multiple computers as opposed to on your local machine, there are additional arguments you may wish to run, but generally the other defaults should be specific.

Pre-processing code

When using the socket approach to parallel processing, each process is started fresh, so things like loaded packages and any variables existing in your current session do not exist. We must instead move those into each process.

The most generic way to do this is the `clusterEvalQ` function, which takes a cluster and any expression, and executes the expression on each process.

```r
clusterEvalQ(cl, 2 + 2)
```

```
[[1]]
[1] 4

[[2]]
[1] 4

[[3]]
[1] 4

[[4]]
[1] 4

[[5]]
[1] 4

[[6]]
[1] 4

[[7]]
[1] 4

[[8]]
[1] 4
```

Note the lack of inheritance:

```r
x <- 1
clusterEvalQ(cl, x)
```

```
Error in checkForRemoteErrors(lapply(cl, recvResult)): 8 nodes produced errors; first error:
```

We could fix this by wrapping the assignment in a `clusterEvalQ` call:

```r
clusterEvalQ(cl, y <- 1)
```

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

[[2]]
[1] 1

[[3]]
[1] 1

[[4]]
[1] 1

[[5]]
[1] 1

[[6]]
[1] 1

[[7]]
[1] 1

[[8]]
[1] 1
```

```r
clusterEvalQ(cl, y)
```

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

[[2]]
[1] 1

[[3]]
[1] 1

[[4]]
[1] 1
```

```
[[5]]
[1] 1

[[6]]
[1] 1

[[7]]
[1] 1

[[8]]
[1] 1
```

```r
y
```

```
Error in eval(expr, envir, enclos): object 'y' not found
```

However, now `y` doesn't exist in the main process. We can instead use `clusterExport` to pass objects to the processes:

```r
clusterExport(cl, "x")
clusterEvalQ(cl, x)
```

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

[[2]]
[1] 1

[[3]]
[1] 1

[[4]]
[1] 1

[[5]]
[1] 1

[[6]]
[1] 1
```

```
[[7]]
[1] 1

[[8]]
[1] 1
```

The second argument is a vector of strings naming the variables to pass.

Finally, we can use `clusterEvalQ` to load packages:

```
clusterEvalQ(cl, {
  library(lme4)
})
```

```
[[1]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[2]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[3]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[4]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[5]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[6]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[7]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"
```

```
[[8]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"
```

Note that this helpfully returns a list of the packages loaded in each process.

Using `par*apply`

There are parallel versions of the three main `apply` statements: `parApply`, `parLapply` and `parSapply` for `apply`, `lapply` and `sapply` respectively. They take an additional argument for the cluster to operate on.

```
parSapply(cl, mtcars, mean)
```

```
       mpg        cyl       disp         hp       drat         wt       qsec
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
        vs         am       gear       carb
  0.437500   0.406250   3.687500   2.812500
```

All the general advice and rules about `par*apply` apply as with the normal `*apply` functions.

Close the cluster

```
stopCluster(cl)
```

This is not fully necessary, but is best practices. If not stopped, the processes continue to run in the background, consuming resources, and any new processes can be slowed or delayed. If you exit R, it should automatically close all processes also. This *does not* delete the `cl` object, just the cluster it refers to in the background.

Keep in mind that closing a cluster is equivalent to quitting R in each; anything saved there is lost and packages will need to be re-loaded.

Continuing the `iris` example

```
cl <- makeCluster(detectCores())
clusterEvalQ(cl, library(lme4))
```

```
[[1]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[2]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[3]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[4]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[5]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[6]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[7]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[8]]
[1] "lme4"      "Matrix"    "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"
```

```r
system.time(save3 <- parLapply(cl, 1:100, f))
```

```
   user  system elapsed 
  0.104   0.009   0.279 
```

```r
stopCluster(cl)
```

Timing this is tricky - if we just time the `parLapply` call we're not capturing the time to open and close the cluster, and if we time the whole thing, we're including the call to lme4. To be completely fair, we need to include loading `lme4` in all three cases. I do this outside of this Markdown file to ensure no added complications. The three pieces of code were, with a complete restart of R after each:

```
reps <- 1000
### lapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save1 <- lapply(seq_len(reps), f)
})

### mclapply
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save2 <- mclapply(seq_len(reps), f)
})


### mclapply
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  cl <- makeCluster(detectCores())
  clusterEvalQ(cl, library(lme4))
  save3 <- parLapply(cl, seq_len(reps), f)
  stopCluster(cl)
})
```

lapply	mclapply	parLapply
5.736	4.353	2.996

Bootstrap example continued

Let's run the bootstrap example with sockets as well.

```r
reps <- 10
system.time({
  cl <- makeCluster(detectCores())
  clusterEvalQ(cl, library(lme4))
  clusterExport(cl, c("recs", "boot"))
  res3 <- parLapply(cl, seq_len(reps), function(x) boot(recs))
  stopCluster(cl)
})
```

```
   user  system elapsed 
  0.466   0.100   8.065 
```

```r
sd3 <- sd(Reduce(c, res3))
c(estcoef + 1.96*sd3, estcoef - 1.96*sd3)
```

```
  typehuq2   typehuq2 
-0.0538258 -0.1710895 
```

In this case, the socket approach is faster than the forking approach - this will not always be the case.