Parallel computing

Tad Dallas

Reading for this week:

Burton, A. W. (1991). Parallel processing in large-scale simulation: motivation and methods. Computing & Control Engineering Journal, 2(2), 74-79.

What is parallel computing?

Parallel computing is the act of using more than a single core of your processor. Most processors have multiple cores (4-core, 8-core, 16-core, etc.). When a process is run, it could occur sequentially (e.g., think of a for loop, going through each index of the loop and iterating forward). The idea of parallel processing is to distribute tasks and data to multiple cores in order to shorten the time it takes to run an analysis (or do the task). Conceptually, think of the for loop, but instead of doing things sequentially, each task is sent to a different CPU core. This type of task, where each task can be broken into a discrete unit, is often called "embarrassingly parallel". I have no idea why, but it is likely somehow linked to the dismissive and toxic culture that permeates tech. It does point to something though. Some tasks can be parallelized, and some cannot. Many tasks can be parallelized at different points, with huge implications to total time for the task. For instance, if you have a model that you would like to run with 1000 parameterizations, and each model is run 10 times, it is possible to parallelize at two different levels. We could parallelize the 1000 parameters, or we could parallelize the model running 10 times. We cannot parallelize both, because it does not work that like that. Thinking about where to parallelize will give you a better idea of what the computer is doing, and how to manage memory and resources.

It may be hard to mentally justify the switch from running something on a single CPU core versus making it amenable to parallel processing, especially when dealing with data sizes and tasks that run fairly quickly. But at times, computations can be:

- cpu-bound: Take too much cpu time
- memory-bound: Take too much memory
- I/O-bound: Take too much time to read/write from disk
- network-bound: Take too much time to transfer

`R` is dumb and has to hold everything in memory, so memory-bound processes are not ideal for parallelization in `R` (there are some circumstances where this is not true). The main constraint that you may run into is processes that are cpu-bound.

Why is it useful?

The time speed up is probably the most useful aspect of parallel computing. At some point in the future, given the increasing availability of data and complexity of models, you will likely encounter a task that you may need to parallelize, or a task that requires too much memory to run on your local machine. In the latter case, you may have access to a cluster, which is essentially a bunch of machines networked together with a decent amount of memory and processing power. But while the memory is available on these machines, you will also have to write the code to be parallelized. Single-core jobs are not really run on clusters, as the power of the cluster really comes from the ability to do many distributed tasks.

How do we do it in R?

So hopefully I have convinced you that parallel processing is a useful skill and is sometimes (oftentimes in my limited experience) essential. So now we want to know how to do it in `R`. There are a number of packages for performing parallel computing tasks, but they mostly boil down to the `parallel` package. This package ships with base `R`, and has functions that may appear really familiar based on earlier lectures. For instance, the `clusterApplyLB` function is an `apply` statement that works on a cluster.

setting up your cluster

Now we will make a cluster. Your local machine almost certainly has more than 1 core. Many phones have more than one core. When you hear things like "quad core CPU", this means that the processor (CPU) has 4 cores. To see how many cores you have access to on your machine, you can use the `detect.cores` function

```
parallel::detectCores()
```

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

The number this returns will depend on whatever machine the document was compiled on. If I am at home, it will likely return "8", as I have a quad-core CPU which is hyperthreaded, so it has 4 physical cores, but 8 virtual cores. If I am on my laptop, it will return "4". If I unnecessarily compile this on the cluster here at Louisiana State University, it might say anywhere between 1 and 64 (or more).

So now we will construct a cluster of 2 nodes. This should hypothetically cut the time our task takes in half, though this ignores the overhead of moving data around to different cores.

```
install.packages('parallel')
```

```
## Installing package into '/home/tad/R/x86_64-pc-linux-gnu-library/3.6'
## (as 'lib' is unspecified)
```

```
## Warning: package 'parallel' is not available (for R version 3.6.3)
```

```
## Warning: package 'parallel' is a base package, and should not be updated
```

```
library(parallel)
```

```
cl <- parallel::makeCluster(2)
cl
```

```
## socket cluster with 2 nodes on host 'localhost'
```

Now we have our cluster. Let us do a simple benchmark test using the `clusterApplyLB` function. The task is to wait `i` seconds where `i` is the vector `c(1,2,3,4,5)`.

```
system.time(
    for(i in 1:5){
        Sys.sleep(i)
    }
)
```

```
##    user  system elapsed
##   0.031   0.017  15.003
```

```
system.time(
    parallel::clusterApplyLB(cl, 1:5, function(x){
        Sys.sleep(x)
    })
)
```

```
##    user  system elapsed
##   0.009   0.027   9.008
```

The first function, which is not parallelized, takes 15 seconds to run. This makes sense, as `sum(1:5)` is 15. Meanwhile, the parallel code takes around half the time, because it can divide the "waiting" between 2 cores. So one core waits for 1 second while the other waits for 2, and as soon as the first core is done waiting one second, it starts to wait 3 seconds. This is called "load balancing", as the individual processes may take different amounts of time. If we were to assume that each parallel process takes the same amount of time, we could enter a situation where some cores are not doing anything while the longer-running processes finishes (this is not ideal).

After we are done with a parallel process, we have to properly shut our cluster down.

```
parallel::stopCluster(cl)
```

Above in the chunk options for this code block, I set `eval=FALSE` so that the `cl` object could still be used by processes below.

Making sure our parallel code is reproducible.

Parallel code is subject to worry about reproducibility just as much as non-parallel code. If we make draws from a distribution, we need to be sure to set the seed with `set.seed()`. In a parallel context, this can be in the function if we have a clear integer index to use. For instance, the code below draws a random uniform variable of length between 1 and 5. We provide the vector `c(1,2,3,4,5)` to the function, so we can use these integers to set the seed.

```
parallel::clusterApplyLB(cl, 1:5,
    function(x){
        set.seed(x)
        runif(x)
    }
)
```

```
## [[1]]
## [1] 0.2655087
##
## [[2]]
## [1] 0.1848823 0.7023740
##
## [[3]]
## [1] 0.1680415 0.8075164 0.3849424
##
## [[4]]
## [1] 0.585800305 0.008945796 0.293739612 0.277374958
##
## [[5]]
## [1] 0.2002145 0.6852186 0.9168758 0.2843995 0.1046501
```

This is a rare exception to what your code will likely look like. If you have a probabilistic component to each process you would like to run in parallel, consider handing the parallel function an index and using the index to specify a row of a data.frame containing relevant parameters. For instance, using the above example, what if we wanted to draw from a uniform distribution, but we wanted a different number of values, a different minimum range, and a different maximum range of the distribution?

```
parms <- data.frame(index=1:5,
    numValues=c(1,2,1,3,5),
    minRange=c(0,1,2,1,0),
```

```
    maxRange=c(5,8,11,8,5))

parallel::clusterExport(cl=cl, c('parms'))

parallel::clusterApplyLB(cl, parms$index,
    function(x){
        set.seed(parms$index)
        runif(parms$numValues[x],
            min=parms$minRange[x],
            max=parms$maxRange[x]
        )
    }
)
```

```
## [[1]]
## [1] 1.327543
##
## [[2]]
## [1] 2.858561 3.604867
##
## [[3]]
## [1] 4.389578
##
## [[4]]
## [1] 2.858561 3.604867 5.009974
##
## [[5]]
## [1] 1.327543 1.860619 2.864267 4.541039 1.008410
```

This would error out, saying that `parms` is not found. But we can use `ls()` and see that parms is in the workspace. This is a common issue. `parms` may be the workspace, but each individual core needs access to the `parms` data.frame, so being in the global workspace does not really matter. This is also true of `R` functions in other packages, so the use of the `require()` function in your code may be necessary to make sure `R` package functionality is accessible across all cores. This is why we use the line:

“‘{r eval=FALSE

parallel::clusterExport(cl=cl, c(‘parms’))

“‘

to make sure that the cluster has access to the data.frame `parms`.

No we have finished with the cluster `cl` and we can stop it using the `parallel::stopCluster` function.

```
parallel::stopCluster(cl)
```

Assorted tips and tricks from someone who fails often

These will likely mostly center around `R`, but are essentially things that I have learned the hard way while programming my way to where I am now.

keep your analytical pipeline organized and consistent between projects

I give all my projects a name in lowerCamelCase, and every project contains the same folders. Keeping this level of organization has allowed me to maintain and push along multiple projects simultaneously. This is not to say that your analytical workflow should not change over time, as it will almost undoubtedly change (and should change). Do not be the equivalent of a person using Excel/SPSS/etc. and refusing to change because it is what you feel most familiar with.

create your own R package

creating an `R` package, either for distribution or just for personal use, is a really rewarding experience. It forces you to document code, handle errors effectively, and write efficient code (since these are functions that will either be released to the public or used by you often enough to be optimized).

use relative paths!

bad: setwd('/home/tad/projectFolder/analysis')

good: setwd('analysis')

force yourself to version your projects

try to mentally remind yourself that every change you make to the project is reason enough to add-commit-push that to Github for proper versioning. This allows you to track your progress, and ensures that you have a backup of your project in case something happens to your local copy.

read more about open source and open access

there are a lot of issues around reproducibility and open science that we did not cover here. Related to reproducibility is the open accessibility of code and data. It is worth reading about open access concepts and approaches to doing science.

consider giving Linux a chance

It is a neat OS that has all the tools that you may have struggled to install and use pretty much out of the box. Plus, it is open source.

consider giving LaTeX a chance

LaTeX is a typesetting language, of which I am a huge fan. It produces beautifully typeset documents, auto-formats references, and handles equations and formatting incredibly well.

write modular code

Do not write the same code twice. Write a general function that can be applied to a diverse set of inputs.

do not put spaces in any file/folder name

Spaces are dumb, and different machines handle them differently. Best to avoid them altogether.

think about data format and legacy issues

Some data formats may last a long time (csv is unlikely to really change), while others are either proprietary (not great for reproducibility) or are now defunct. Keeping your data in a format that is as stable as possible goes a long way to helping keep your analyses reproducible. xls is a terrible format for data.