Parallel Computing in R

Introduction to Computational BioStatistics with R

Alexey Fedoseev

November 26, 2024

Institute of Medical Science
UNIVERSITY OF TORONTO

Concurrency vs Parallelism

Figure 1: Concurrent, non-parallel execution

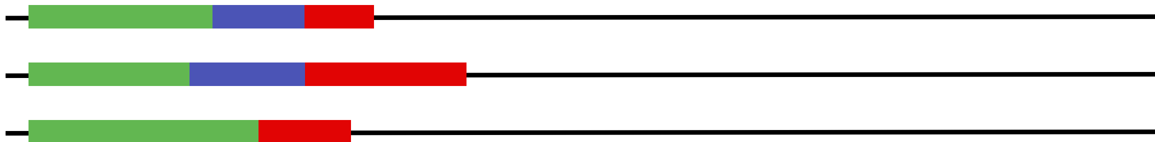


Figure 2: Concurrent, parallel execution

Using multiple processors in R

In this class we will cover using multiple processors and/or nodes to do large-scale computations in R.

- existing parallelism
- `parallel` package:
  - `multicore` (use all cores on a computer): non-windows
  - `snow` (use all cores on a computer, or across a cluster)

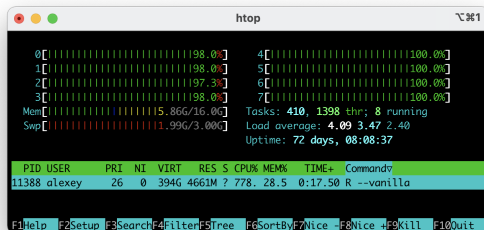
Existing parallelism

It's important to realize that many fundamental routines as well as higher-level packages come with some degree of scalability and parallelism "baked in".

Open another terminal to your node, and run "top" while executing the following in R:

```
>
> n <- 4 * 1024
>
> A <- matrix( rnorm(n * n), ncol = n, nrow = n )
> B <- matrix( rnorm(n * n), ncol = n, nrow = n )
>
> C <- A %*% B
>
```

Existing parallelism



One R process using 778% of a processor.

R can (and should) be built using high-performance threaded libraries for math in general, and linear algebra in particular.

Here the single R process has launched several threads of execution – all of which are part of the same process, and so can see the same memory.

Packages that explicitly use parallelism

For a complete list, see

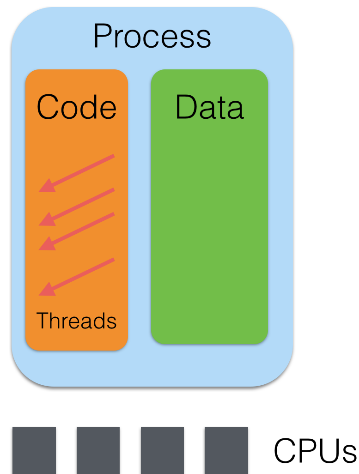
http://cran.r-project.org/web/views/HighPerformanceComputing.html

Plus packages that use linear algebra or other expensive math operations which can be implicitly multithreaded.

When at all possible, don't do the hard work yourself — look to see if a package already exists which will do your analysis at scale.

Processes vs Threads vs CPUs vs Cores

- A process is a running program. It has data, the program code, and one or more threads of execution - points in the code that is currently being run.
- A thread can see all of the data (and all other threads) within a process; you can't see anything outside of your own user process.
- The operating system assigns running threads to cores (or CPUs, or processors, which are the same thing and I'll use the terms interchangably.)
- "Core" is the least ambiguous term — an independent processing unit.



The `parallel` Package

Since R 2.14.0 (late 2011), the `parallel` package has been part of core R. It incorporates - and mostly supersedes - two other packages:

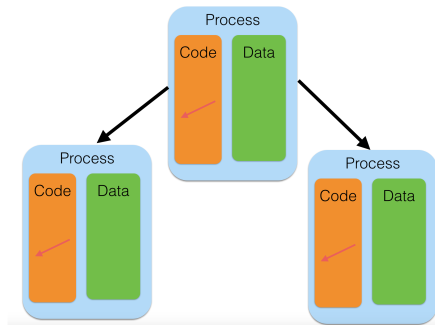
- `multicore`: for using all cores on a single processor. Not on Windows
- `snow`: for using any group of processors, possibly across a cluster

Many packages which use parallelism use one of these two, so it is worth understanding.

Both create new processes (not threads) to run on different processors; but differ in important ways.

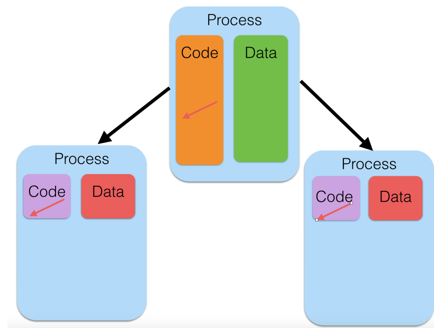
`multicore` - forking

- `multicore` creates new processes by forking — cloning – the original process.
- That means the new processes start off seeing a copy of exactly the same data as the original. If a first process can read a file, and it then forks two new processes - each will see a copy of the file.
- These are not shared memory; changes in one process will not be reflected in others.
- Windows doesn't have fork(), so windows can't use these routines.



snow - Spawning

- snow creates entirely new R processes to run the jobs.
- A downside is that you need to explicitly copy over any needed data, functions.
- But the upsides are that spawning a new process can be done on a remote machine, not just current machine. So you can in principle use entire clusters.
- In addition, the flipside of the downside: new processes don't have any unneeded data
  - less total memory footprint.



mcparallel/mccollect

The simplest use of the `multicore` package is the pair of functions `mcparallel()` and `mccollect()`. `mcparallel()` forks a task to run a given function; it then runs in the background. `mccollect()` waits for and gets the result.

```
> sleep <- function(t) {
+   Sys.sleep(t) # time in seconds
+   return(t)
+ }
> system.time(sleep(10))
   user  system elapsed
  0.000   0.000  10.005
>
> par.sleep <- function() {
+   sleep20 <- mcparallel(sleep(20))
+   sleep30 <- mcparallel(sleep(30))
+   return(mccollect(list(sleep20, sleep30)))
+ }
```

mcparallel/mccollect

```
> system.time(ans <- par.sleep())
   user  system elapsed
  0.005   0.006  30.009
> ans
$`58550`
[1] 20

$`58551`
[1] 30
```

We get a list of answers, with each element “named” by the process ID that ran the job.

mcparallel/mccollect

When parallelizing your code, make sure you have enough work for all your processes.

Let us say we have a serial and a parallel version of the same code:

```
> mean.rnd <- function(nmax = 1e7) {
+   fst.mean <- mean(rnorm(nmax))
+   snd.mean <- mean(rnorm(nmax))
+   return(list(fst.mean, snd.mean))
+ }
>
> par.mean.rnd <- function(nmax = 1e7) {
+   fst.mean <- mcparallel(mean(rnorm(nmax)))
+   snd.mean <- mcparallel(mean(rnorm(nmax)))
+   return(mccollect(list(fst.mean, snd.mean)))
+ }
```

mcparallel/mccollect

If we run these function, but there is not enough work for each process, the parallel version end up running longer than our serial code:

```
> system.time(mean.rnd(1e4))
   user  system elapsed
  0.001   0.000   0.001
>
> system.time(par.mean.rnd(1e4))
   user  system elapsed
  0.002   0.004   0.005
```

Forking the processes and waiting for them to rejoin takes some time. This overhead means that we want to launch jobs that take a significant length of time to run - much longer than the overhead (hundredths to tenths of seconds for fork()).

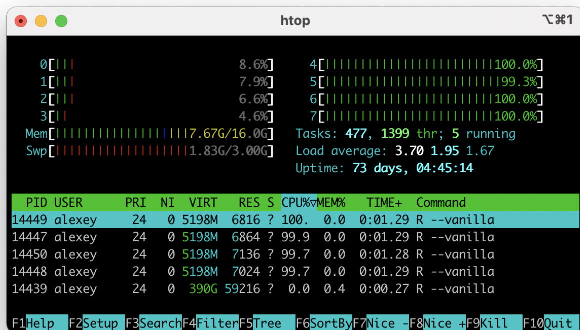
mclapply

Another way to fork code is to use `mclapply`, which works the same way as `lapply`, but forking off the processes (as with `mcparallel`):

```
> add.me <- function(n) {
+   a <- 1
+   for (i in 1:n) a <- 1 / (1 + a)
+ }
> system.time(list.res <- lapply(rep(1e8,4), add.me))
   user  system elapsed
  5.532   0.007   5.539
> system.time(list.par.res <- mclapply(rep(1e8,4), add.me, mc.cores = 4))
   user  system elapsed
  4.495   0.013   1.516
```

mclapply

Note what the output of top looks like when this is running:



There are multiple processes running - not one process using multiple CPUs via threads.

Parallel RNG

Depending on what you are doing, it may be very important to have different (or the same!) random numbers generated in each process.

`parallel` has a good RNG suitable for parallel work based on the work of Pierre L'Ecuyer in Montréal:

```
> RNGkind("L'Ecuyer-CMRG")
> mclapply(rep(1,2), rnorm, mc.cores=2, mc.set.seed=TRUE)
[[1]]
[1] -0.4982475

[[2]]
[1] 0.9267458
```

pvec - simplified mclapply

For the simple and common case of applying a function to each element of a vector and returning a vector, the parallel package has a simplified version of `mclapply` called `pvec`.

```
> fx <- function(x) {return(x^5-x^3+x^2-1)}
> maxn <- 1e7
>
> system.time( res <- sapply(1:maxn, fx) )
   user  system elapsed
  5.979   0.554   6.558
>
> system.time( res <- pvec(1:maxn, fx, mc.cores=4) )
   user  system elapsed
  0.359   0.122   0.225
```

Multiple computers with parallel/snow

The snow package allows us to launch new R processes - by default, on the current computer, but also on any computer you have access to (SNOW stands for "Simple Network of Workstations", which was the original use case).

The recipe for doing computations with snow looks something like:

```
library(parallel)
cl <- makeCluster(nworkers,...)
results1 <- clusterApply(cl, ...)
results2 <- clusterApply(cl, ...)
stopCluster(cl)
```

other than the makeCluster()/stopCluster(), it looks very much like multicore and mclapply.

Hello world

Let's try starting up a "cluster" (eg, a set of workers) and generating some random numbers from each:

```
> library(parallel)
> cl <- makeCluster(3)
> clusterCall(cl, rnorm, 5)
[[1]]
[1]  0.5341856  1.1630899 -1.4428501  2.2743744 -0.8053170

[[2]]
[1]  0.4182711 -0.6244335  1.7348715  0.8064860 -0.4366544

[[3]]
[1] -0.4195192 -0.7158211 -0.2013389 -0.7199361  0.9807875

> stopCluster(cl)
```

Clustering on Clusters

Let us assume that we have several datasets, variables and functions loaded in our workspace:

```
> rnorm.fx <- function(n) {return(rnorm(n))}
> sum.rnorm <- function(n) {return(sum(rnorm.fx(n)))}
```

Recall that we aren't forking here; we are creating processes from scratch. These processes, new to this world, are not familiar with our workspace.

```
> cl <- makeCluster(3)
> clusterCall(cl, sum.rnorm, 5)
Error in checkForRemoteErrors(lapply(cl, recvResult)) :
  3 nodes produced errors; first error: could not find function "rnorm.fx"
```

Clustering on Clusters

We actually have to ship the data out to the workers:

```
> clusterExport(cl, "rnorm.fx")
> clusterCall(cl, sum.rnorm, 5)
[[1]]
[1] 0.6790298

[[2]]
[1] -5.254114

[[3]]
[1] -0.09843904

> stopCluster(cl)
```

Note that the costs of shipping out data back and forth, and creating the processes from scratch, is relatively costly - but this is the price we pay for being able to spawn the processes anywhere.

Running across machines

You can run your code on multiple machines by specifying them to the function `makePSOCKcluster`:

```
hosts <- c( rep("localhost", 4) )
cl <- makePSOCKcluster(names=hosts)
clusterCall(cl, rnorm, 5)
clusterCall(cl, system, "hostname")
stopCluster(cl)
```

clusterApply

`clusterApply` takes a cluster, a sequence of arguments (can be a vector or a list), and a function, and calls the function with the first element of the list on the first node, with the second element of the list on the second node, and so on, recycling nodes as needed.

```
> clusterApply(cl, 1:2, sum, 3)
[[1]]
[1] 4

[[2]]
[1] 5
```

Summary: parallel/snow

The `cluster` routines in `parallel` are good if you know you will eventually have to move to using multiple computers (nodes in a cluster, or desktops in a lab) for a single computation.

- Use `clusterExport` for functions and data that will be needed by everyone
- Communicating data is slow, but *much* faster than having every worker read the same data from a file
- Use `clusterApplyLB` if the tasks vary greatly in runtime
- Use `clusterApply` if each task requires an enormous amount of data
- Use `makePSOCKcluster` for clusters