Parallelization in R

Xunzhao Yin

September 11, 2017

Caution: don't simply copy & paste the following code. For some font problems, it won't work. Retyping is a good idea.

```
N <- 5e8 # Half billion. Might drain your laptop's memory.
M <- 10; V <- 4
Alpha <- 2 + M^2 / V; Beta <- M * (Alpha - 1)
system.time(MCMCpack::rinvgamma(N, Alpha, Beta)) # system.time
   checks how long it takes to finish the job.
```

The above code generates half billion random numbers from an inverse-Gamma distribution. It takes 41 seconds:

```
##   user  system elapsed
## 40.461   0.785  41.326
```

And when I check the CPU usage, I find 7 cores are actually idling and the rest 1 is working at its 100% capacity. The job, which is "perfectly parallel", could have been done faster if all 8 cores could work together.

Mayby we can divide the job into 8. If we do that, maybe R will run these 8 small jobs separately on 8 different cores. Let's try.

```
  RanIG <- function(x) MCMCpack::rinvgamma(N / 8, Alpha, Beta)
  system.time(lapply(1:8, RanIG))
```

Note that a `lapply` function is used instead of a `for` loop. Their difference lies in the observation that a `for` loop is "inherently serial": the next job will not begin until the previous one is finished. But a `lapply` handles all 8 jobs together and therefore is potentially faster. However, it turns out that R did all 8 jobs on one core, and leave the other 7 idle. How stupid!

```
##    user  system elapsed
##  41.751   0.856  42.679
```

1 Parallelization on a Multi-core Laptop

1.1 The `parallel` Package

1.1.1 Resources

1. Max Gordon - How to go parallel in R (Click me).

2. Josh Errickson - Parallel Processing in R (Click me).

1.1.2 How it Works?

The `parallel` package is arguably the most widely-used package for parallel processing in R. You might have heard of other packages like `multicore` or `snow`. Actually, `parallel` combines those two into one. Its functions are either from `multicore` ore `snow`. To load the package,

```
  library(parallel)
```

But the package does not automatically give your R the ability of parallel computation. You need to group your cores into a group, or called, a *cluster*.

```
  NoCores <- detectCores() - 1 # Check the number of cores you
     have on your laptop. And leave one idle.
  cl <- makeCluster(NoCores, type = "FORK") # Set up a cluster.
```

(If an error occurs in R when trying the above code on your own computer, please refer to Section 1.1.3 for a solution.) My laptop has 8 cores. So NoCores = 7. Now let's divide the job into 7, and try the parallel computation. The counterpart of `lapply` in Package `parallel` is `parLapply`.

```
RanIG <- function(x) MCMCpack::rinvgamma(N / NoCores, Alpha, Beta
   )
clusterSetRNGStream(cl)
```

```
system.time(parLapply(cl, 1:NoCores, RanIG))
stopCluster(cl)
##   user  system elapsed
##  0.730   2.819  14.615
```

Note that it is NOT as fast as I thought it could be. I thought the parallel computation with 7 cores would be 7 times as fast as before. But actually it's only 3 times as fast.

Why is it not as fast as we thought? The reason lies in something called "overhead". Intuitively, transferring data between cores takes time. So going parallel won't improve the speed if your job is small.

1.1.3 Problem 1: FORK or PSOCK

If you are running R on Windows, copying & pasting the above code won't work. Why?

There are two ways in which a cluster can be set up. They function differently:

```
cl <- makeCluster(NoCores, type = "FORK")
```

or

```
cl <- makeCluster(NoCores, type = "PSOCK")
```

Type PSOCK works on all operation systems, including Windows, while Type FORK works on all operation systems but Windows. Therefore, on a Windows system, you need to code this way:

```
library(parallel)
RanIG <- function(x) MCMCpack::rinvgamma(N / NoCores, Alpha, Beta
   )
cl<-makeCluster(NoCores, type = "PSOCK")
clusterExport(cl, c("N", "NoCores", "Alpha", "Beta", "RanIG"))
system.time(parLapply(cl, 1:NoCores, RanIG))
```

```
stopCluster(cl)
```

Note that on a PSOCK cluster, you need the `exportcluster` function to load variables or functions into each core. Therefore, unsurprisingly a PSOCK cluster is slower than a FORK one:

```
##    user  system elapsed
##   4.230   2.531  18.769
```

The differences, as well as pros and cons, of the two types of cluster are briefly discussed in Josh Errickson's Parallel Processing in R (Click me).

1.1.4 Problem 2: Random Number Generator

Type FORK is the default cluster type on a Linux or Mac OS system. It's easier to code, and it runs faster than a Type PSOCK cluster. However, it also has its problems:

```
N <- 7
library(parallel)
NoCores <- detectCores() - 1
cl <- makeCluster(NoCores, type = "FORK")
parSapply(cl, 1:7, function(x) rnorm(1)) # parSapply is similar
    to parLapply. The former wraps the results in a vector.
stopCluster(cl)
```

The above code groups 7 cores into a cluster. We want it to random draw 7 numbers from a standard normal distribution. However, the result is as follows:

```
## [1] 0.3113058 0.3113058 0.3113058 0.3113058 0.3113058
    0.3113058 0.3113058
```

Seven numbers are exactly the same. You would feel this is ridiculous. We thought the 7 cores run independently from each other. But it turns out that under a FORK cluster, the 7 cores share a same random number generator.

How to solve the RNG problem? Two solutions. The first one is to abandon the Type FORK and substitute with a Type PSOCK:

```
N <- 7
library(parallel)
NoCores <- detectCores() - 1
cl <- makeCluster(NoCores, type = "PSOCK")
clusterExport(cl, c("N", "NoCores"))
parSapply(cl, 1:7, function(x) rnorm(1))
## [1] -0.31725379  0.38989402  0.03233058 -0.30088285
## [5] 0.43269750  0.16416976  -1.38538207
stopCluster(cl)
```

And the other solution is that, if you really want to stay with Type FORK, use function `clusterSetRNGStream`:

```
N <- 7
library(parallel)
NoCores <- detectCores() - 1
cl <- makeCluster(NoCores, type = "FORK")
clusterSetRNGStream(cl)
parSapply(cl, 1:7, function(x) rnorm(1))
## [1]  1.1455837 -0.6943564  1.1800749  0.3537022 -0.3516522
## [6]  1.5184780  1.7604563
stopCluster(cl)
```

1.2 The `foreach` Package

1.2.1 Resources

1. Max Gordon - How to go parallel in R (Click me).
2. Steve Weston - Using the `foreach` Package (Click me).
3. Steve Weston and Rich Calaway - Getting Started with `doParallel` and `foreach` (Click me).

1.2.2 How It Works?

Package `foreach` defines a new loop, which is basically a hybrid of the `for` loop with the family of `apply` loops. We mentioned that the `for` loop is inherently serial. In contrast, the `foreach` loop supports parallel computation and is easy to use. Please refer to *Steve Weston - Using the `foreach` Package* (Click me) for more details.

Following is an example to illustrate how the new loop works. Still, we want to randomly draw half billion numbers from an inverse Gamma distribution. We go parallel and want to know how long the computation takes. A FORK cluster is set up.

```
N <- 5e8
M <- 10; V <- 4
Alpha <- 2 + M^2 / V; Beta <- M * (Alpha - 1)
NoCores <- parallel::detectCores() - 1

library(foreach)
library(doParallel)

cl<-makeCluster(NoCores, type = "FORK")
registerDoParallel(cl)
clusterSetRNGStream(cl)
system.time(foreach (i = 1:NoCores, .packages = "MCMCpack")
%dopar% rinvgamma(N / NoCores, Alpha, Beta))
stopCluster(cl)
```

The result is as follows. It seems that the `foreach` loop is a little bit slower than the `parLapply` illustrated in Section 1.1.2.

```
##    user  system elapsed
##   0.622   2.842  16.436
```

Note that, compared with Section 1.1.2, a `registerDoParallel` step is needed in order for the `foreach` to work.

And if you are on a Windows system, or would like to use a PSOCK cluster, you can symply change ”FORK” into ”PSOCK”, and nothing else needs to be changed. The result is as follows. Again, we see a PSOCK cluster is slower than a FORK one.

```
> N <- 5e8
> M <- 10; V <- 4
> Alpha <- 2 + M^2 / V; Beta <- M * (Alpha - 1)
> NoCores <- parallel::detectCores() - 1
>
> library(foreach)
> library(doParallel)
>
> cl<-makeCluster(NoCores, type = "PSOCK")
> registerDoParallel(cl)
> clusterSetRNGStream(cl)
> system.time(foreach (i = 1:NoCores, .packages = "MCMCpack") %
   dopar% rinvgamma(N / NoCores, Alpha, Beta))
##   user  system elapsed
##  4.380   2.505  19.620
> stopCluster(cl)
```

2 KU Cluster

The Center for Research Computing has a cluster with over 24,000 processing cores. It's actually thousands of computers called *node* bundled together. Each node has 20 cores.

Students of the College of Liberal Art and Science can use the cluster for free. But you need to apply for an account.

2.1 How to Apply for an Account

This is the website to apply for a new account (Click me).

2.2 How to Access the Cluster

Click me for a tutorial about how to access the cluster.

The CRC cluster is a net work of thousands of Linux computers. Therefore, if you are using a Linux system on your laptop, you will find the cluster extremely easy to access. And when using the cluster, you feel it just the same as working on your own laptop.

If you are on a Mac OS system and you know how to use a Terminal, you will find the cluster easy to use.

If you are on a Windows system, refer to the website above. The CRC provides a GUI software through which you submit your jobs to the cluster. And you will get the results back after the job is finished. You do not have such freedom as you have on a Linux system, but it stills works. But I strongly suggest you to install a Linux system on you laptop.

2.3 How it works on the Cluster

I will request 1 node with 20 cores from the CRC cluster. Then draw half billion random numbers from a inverse Gamma distribution with the `foreach` loop with a type FORK, just as we did in Section 1.2.2.

2.3.1 How to login

After you get an account, type in your Terminal

```
ssh -X username@hpc.crc.ku.edu
```

and put in your passcode.

2.3.2 How to Request Nodes and Cores

In your terminal, type

```
msub -X -I -l nodes=1:ppn=20
```

2.3.3 How to Launch R on the Node

Run the following two commands

```
module load R
R
```

R on the node has no graphical user interface. Actually, after running the above command line, your Terminal becomes your R.

2.3.4 How Long does the Job Takes

```
##    user  system elapsed
##   0.450   3.141   9.335
```

You are actually using that computer in CRC. You don't have to worry about the memory of your own laptop. You can do something else on your laptop while the node remote is doing the job.

3 In Summary

At the beginning, the job takes more than 40 seconds. Eventually, it can be done within 10 seconds if we parallelize it on the CRC cluster.