Parallel Computation

Biostatistics 140.776

Parallel Computing

- Parallel computation is the simultaneous execution of different pieces of a larger computation across multiple computing processors or cores.
- Many computations in R can be made faster by the use of parallel computation.
- The basic idea is that if you can execute a computation in X seconds on a single processor, then you should be able to execute it in X/n seconds on n processors.
- Such a speed-up is generally not possible because of overhead and various barriers to splitting up a problem into n pieces, but it is often possible to come close in simple problems.

Parallel Computing

- It used to be that parallel computation was "high-performance computing", where expensive machines were linked together via high-speed networking to create large clusters of computers.
- There you needed sophisticated software to manage the communication of data between different computers in the cluster.
- Parallel computing in that setting was a highly tuned, and carefully customized operation
- Almost all computers now contain multiple processors or cores on them (even your phone likely comes with a dual-core CPU).
- We will focus on functions that can be used on **multi-core computers**, which these days is almost all computers.

You May Already be Doing it

Many software packages are linked with computational libraries that operate in parallel

- R uses BLAS for linear algebra computation, which may use a parallel implementation
- Mac: Accelerate Framework
- Other: ACML, Intel MKL
- Automatically Tuned Linear Algebra Software (ATLAS)

You may need to compile R from sources to use these libraries.

What Can We Parallelize?

- Any process/job that can be split into multiple **independent** pieces may be paralleized
- Each of the pieces should be roughly similar in size/difficulty (evenly split)
- The bootstrap! Each bootstrap resample is independent of the other
- "Embarrassingly parallel" computing

Embarrassing Parallelism

It's easy to think about how to execute an embarrasingly parallel computation. Think of how the `lapply()` function works. The `lapply()` function requires

1. A list, or an object that can be coerced to a list.
2. A function to be applied to each element of the list
3. `lapply()` applies that function to each element of the list

Finally, recall that `lapply()` always returns a list whose length is equal to the length of the input list.

Embarrassing Parallelism

- The `lapply()` function works much like a loop–it cycles through each element of the list and applies the supplied function to that element.
- While `lapply()` is applying your function to a list element, the other elements of the list are just. . . sitting around in memory.
- The order in which elements are processed is not important
- The function being applied to a given list element does not need to know about other list elements.
- Almost any operation that can be done with `lapply()` can be parallelized.

Parallel `lapply()`

A list object can be split across multiple cores of a processor and then a function can be applied to each element of the list object on each of the cores.
Conceptually, the steps in the parallel procedure are

1. Split list `X` across multiple cores
2. Copy the supplied function (and associated environment) to each of the cores
3. Apply the supplied function to each subset of the list `X` on each of the cores in parallel
4. Assemble the results of all the function evaluations into a single list and return

Methods for Parallel Computing in R

Forking

- The parent R process is "forked" into multiple child processes, one for each core, where some piece of the work is done
- Communication is done through inter-process communication and shared memory
- Generally only available on Unix-like system

Sockets

- Multiple instances of R are started up separately
- Each process is independent with separate memory
- Communication done via *sockets*
- Available on all systems

The `parallel` Package

Key functions in the `parallel` package are

- `mclapply()`: a parallel version of `lapply()` using forking that works in much the same way as `lapply()`
- `parLapply()`: a parallel version of `lapply()` using sockets (requires some additional setup beforehand)

`mclapply()`

The `mclapply()` function works much like `lapply()`

- First argument is a list or an object that can be converted to a list
- Second argument is a function to be applied to each element (the first argument is an element of the list)
- `...`: other arguments for your function
- `mc.cores`: the number of cores to split your problem over
-
  - It is possible to set `mc.cores` to $>=$ the number of cores you have and you may get some extra speed—but not predictable (remember your computer still has other things to do!)

How Many Cores?

The first thing you might want to check with the `parallel` package is if your computer in fact has multiple cores that you can take advantage of.

```
library(parallel)
detectCores()
```

```
[1] 4
```

This shows the number of *logical* cores by default.
On some systems you can call `detectCores(logical = FALSE)` to return the number of physical cores.

```
detectCores(logical = FALSE)
```

```
[1] 2
```

Using `mclapply()`

An exmaple of splitting a job across 10 cores

```
r <- mclapply(1:10, function(i) {
        Sys.sleep(10)  ## Do nothing for 10 seconds
}, mc.cores = 10)
```

Using `mclapply()`

```
Processes: 305 total, 3 running, 16 stuck, 286 sleeping, 1275 threads                                1
Load Avg: 1.62, 1.29, 1.09  CPU usage: 4.45% user, 0.40% sys, 95.13% idle
SharedLibs: 269M resident, 22M data, 35M linkedit.
MemRegions: 51208 total, 5946M resident, 140M private, 1358M shared.
PhysMem: 15G used (3723M wired), 49G unused.
VM: 889G vsize, 529M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 2133108/2635M in, 1073520/148M out. Disks: 3040748/50G read, 1657520/59G w

PID    COMMAND      %CPU  TIME     #TH    #WQ  #PORT MEM    PURG   CMPR PGRP  PPID  STATE
22412  f951         100.2 00:06.24 1/1    0    12    303M+  0B     0B   94358 22411 running
89401  RStudio      5.7   01:07.60 44     23   331   369M+  4620K  0B   89401 1     sleeping
94299  top          2.5   00:04.97 1/1    0    44    3756K+ 0B     0B   94299 72678 running
193    WindowServer 1.7   04:16.86 4      1    393   126M   4328K  0B   193   1     sleeping
1050   Terminal     1.1   02:55.46 29     23   323   67M    2732K  0B   1050  1     sleeping
0      kernel_task  0.9   12:50.30 195/24 0    2     3117M  0B     0B   0     0     running
22414  rsession     0.2   00:00.00 1      0    7     3172K  0B     0B   89401 89403 sleeping
22421  rsession     0.2   00:00.00 1      0    7     3028K  0B     0B   89401 89403 sleeping
22420  rsession     0.2   00:00.00 1      0    7     3100K  0B     0B   89401 89403 sleeping
22422  rsession     0.2   00:00.00 1      0    7     3648K  0B     0B   89401 89403 sleeping
22415  rsession     0.2   00:00.00 1      0    7     3196K  0B     0B   89401 89403 sleeping
22419  rsession     0.2   00:00.00 1      0    7     3028K  0B     0B   89401 89403 sleeping
22416  rsession     0.2   00:00.00 1      0    7     3152K  0B     0B   89401 89403 sleeping
22413  rsession     0.2   00:00.00 1      0    7     3184K  0B     0B   89401 89403 sleeping
22417  rsession     0.1   00:00.00 1      0    7     3112K  0B     0B   89401 89403 sleeping
22418  rsession     0.1   00:00.00 1      0    7     3076K  0B     0B   89401 89403 sleeping
89403  rsession     0.0   00:05.64 7      0    40    56M    0B     0B   89401 89401 sleeping
```

Figure 1: Multiple sub-processes spawned by `mclapply()`

Using `mclapply()`

Another example, using real data.

```
library(readr)
infiles <- dir("specdata", full.names = TRUE)
specdata <- lapply(infiles, read.csv)
```

The `specdata` object is a list of data frames, with each data frame corresponding to each of the 332 monitors in the dataset.

Computing Summary Statistics

One thing we might want to do is compute a summary statistic across each of the monitors.

- We might want to compute the percentiles of sulfate for each of the monitors.

The serial version:

```
system.time({
        mn <- lapply(specdata, function(df) {
                quantile(df$sulfate, seq(0, 1, .1),
                        na.rm = TRUE)
        })
})
```

```
   user  system elapsed
  0.358   0.052   0.411
```

Computing Summary Statistics

The equivalent call using `mclapply()` would be

```
system.time({
        mn <- mclapply(specdata, function(df) {
                quantile(df$sulfate, seq(0, 1, .1),
                         na.rm = TRUE)
        }, mc.cores = 12)
})
```

```
   user  system elapsed
  0.140   0.127   0.107
```

Watch Out!

- One advantage of serial computations is that it allows you to better keep a handle on how much **memory** your R job is using.
- When executing parallel jobs via `mclapply()` it's important to pre-calculate how much memory *all* of the processes will require
- Make sure this is **less than** the total amount of memory on your computer.

Error Handling

- When `mclapply()` is called, the functions supplied will be run in the sub-process while effectively being wrapped in a call to `try()`
- This allows for one of the sub-processes to fail without disrupting the entire call to `mclapply()`
- If one sub-process fails, it may be that all of the others work just fine and produce good results.
- This error handling behavior is a significant difference from the usual call to `lapply()`
- With `mclapply()`, when a sub-process fails, the return value for that sub-process will be an R object that inherits from the class `"try-error"`

Error Handling

The code below deliberately causes an error in the 3 element of the list.

```
r <- mclapply(1:5, function(i) {
        if(i == 3L)
                stop("error in this process!")
        else
                return("success!")
}, mc.cores = 5)
```

```
Warning in mclapply(1:5, function(i) {: scheduled cores 3 e
errors in user code, all values of the jobs will be affecte
```

Here we see there was a warning but no error in the running of the above code. We can check the return value.

Error Handling

Looking at the returned list object

```
str(r)
```

```
List of 5
 $ : chr "success!"
 $ : chr "success!"
 $ :Class 'try-error'  atomic [1:1] Error in FUN(X[[i]], .

  .. ..- attr(*, "condition")=List of 2
  .. .. ..$ message: chr "error in this process!"
  .. .. ..$ call   : language FUN(X[[i]], ...)
  .. .. ..- attr(*, "class")= chr [1:3] "simpleError" "erro
 $ : chr "success!"
 $ : chr "success!"
```

Note that the 3rd list element in `r` is different.

What Can't You Parallelize?

Loops where things depend on computed values from other elements are difficult to parallelize

```
x <- numeric(10)
x[1] <- 0  ## Initialize
for(i in 2:10) {
        x[i] <- 0.7 * x[i-1] + rnorm(1)
}
```

Iterative optimization algorithms (EM algorithm, Newton's method, MCMC) are also difficult to parallelize because of dependencies

Building a Socket Cluster

- Using the forking mechanism on your computer is one way to execute parallel computation but it's not the only way that the `parallel` package offers.
- Another way to build a "cluster" using the multiple cores on your computer is via *sockets*.
- A socket is simply a mechanism with which multiple processes or applications running on your computer (or different computers, for that matter) can communicate with each other.
- With parallel computation, data and results need to be passed back and forth between the parent and child processes and sockets can be used for that purpose.

Socket Clusters

Building a socket cluster is simple to do in R with the `makeCluster()` function.

```
library(parallel)
cl <- makeCluster(12)
```

The `cl` object is an abstraction of the entire cluster and is what we'll use to indicate to the various cluster functions that we want to do parallel computation.

Socket Cluster

To do an `lapply()` operation over a socket cluster we can use the `parLapply()` function.

```
x <- rnorm(1)
power <- parLapply(cl, 1:10, function(num) {
        x^num
})
```

```
Error in checkForRemoteErrors(val): 10 nodes produced erro
```

This won't work because the other cluster processes do not know about `x`.

Exporting Data

Any data orother information that the child process will need to execute your code, needs to be **exported** to the child process from the parent process via the `clusterExport()` function.

- This behavior is the key difference from `mclapply()`

```
clusterExport(cl, "x")
parLapply(cl, 1:10, function(num) {
        x^num
})
```

```
[[1]]
[1] 0.5977223

[[2]]
[1] 0.357272

[[3]]
[1] 0.2135494
```

Summary

- Most computers these days have multiple processors or cores, allowing for parallel computing
- Embarrassingly parallel computations are common in many data applications
- Almost any process using `lapply()` can be parallelized
- `mclapply()` uses the forking mechanism to split work across processes
- `parLapply()` uses sockets/multiple processes to split work; generalizes to multiple computers
- Can't parallelize computations with critical dependencies