Parallel Processing

December 24, 2017

Parallel processing is a technique that can be used to make programs run faster by running multiple calculations simultaneously on multiple cores. It typically involves splitting up a data set into smaller pieces, concurrently processing each piece on a separate core, and then putting the results back together. The original process is known as the master node, and the others are known as slave nodes. Not every problem is easily parallelized, but for those that can be parallelized, this technique can significantly speed up the execution of one's program.

There are several packages in R for doing parallel processing. Currently, the most popular is the `parallel` package, which we will illustrate here. For our example, we will consider calculating the mean of a large vector of numbers. This is the kind of problem that can be easily parallelized. We simply chop up our vector into smaller vectors, calculate the sum of each vector, and then sum the results and divide the by length of the original vector. In R, this can be done as follows:

```
library(parallel)

parallel_mean <- function(x) {

  cluster <- makeCluster(detectCores() - 1)

  on.exit(stopCluster(cluster))

  sums <- parSapply(cluster, x, sum)

  sum(sums)/length(x)

}

x <- 1:50000

print(parallel_mean(x))
```

```
## [1] 25000.5
```

Let's now explain what this does. `makeCluster` creates the nodes which are used to execute each sub-problem. This function supports two methods of clustering: sockets and forks. The socket method spawns a new, fresh instance of R on each node, and slave nodes communicate with the master node via network sockets. Under the fork method, the current R instance is copied to each node. The default is the socket method. Forks are usually faster and are easier from a programming perspective, but unfortunately they are not supported on Windows, so we use sockets here, and we will not be covering the forking method. The first argument to `makeCluster` is the number of nodes, which we set to `detectCores() - 1`. `detectCores` returns the number of available nodes. Personally, I usually use all but one available node, particularly if running the program on my own computer. This leaves one node free for me to use an editor, check email, etc. These things can still be done otherwise since your program does not have exclusive use of the cores, but you may find your computer fairly unresponsive if your program uses all cores.

`stopCluster` shuts down the worker nodes. It is safest to put it inside a call to `on.exit`, a function which executes an expression upon function exit. This assures that the `stopCluster` command always executes, even if an error occurs within the function.

Warning: Only use `on.exit` within a function. If `on.exit` is called from global scope, the expression will never be evaluated since there is no function to return (in a perfect world, this would produce a warning, but

we don't live in a perfect world).

`parSapply` is parallel version of 'sapply'. Similarly, `parLapply`, `parApply`, and `clusterMap` are parallel versions of `lapply`, `apply`, and `mapply`, respectively. These are the functions that do most of the work in the `parallel` package. They automatically split the first argument into a number of pieces, send the pieces to each node for processing, and return the results. There are also "load-balancing" versions of these functions, which try to assure that each node has roughly the same amount of work. Load balancing can improve performance in some cases, but performance can be worse in other cases.

So, did we actually make things faster? Let's see.

```
# Now let's time things
{

  start <- proc.time()
  replicate(5, parallel_mean(x))
  end <- proc.time()

  print(end - start)

}
```

```
##    user  system elapsed
##    0.19    0.11    5.86
```

```
{

  start <- proc.time()
  replicate(5, mean(x))
  end <- proc.time()

  print(end - start)

}
```

```
##    user  system elapsed
##       0       0       0
```

Uh-oh! The parallel version is actually slower. This is because there is overhead in creating the nodes, dividing the work, and communicating results to the master node. For parallel processing to be faster, we need to provide enough work to the nodes to make up for the overhead. Averaging a few thousand numbers just doesn't cut it.

Other useful functions

If using the socket method, any variable defined after the cluster is created is only available to the master node. To make it available to all nodes, one can use the `clusterExport` function. Also, to run a block of R code on all nodes, one must pass the block to a function called `clusterEvalQ`. For example, to make a library available on all nodes, one would do the following:

```
clusterEvalQ(cluster, {

  library(dplyr)
  library(ggplot2)

})
```