Post2 - Optimized Programming

Haoming Yang

11/25/2017

Introduction:

This post mainly focus on optimize study. By ‘optimize’, it means how to finish same task in a shorter period, or we can simply understand it as speedup.

Motivation:

This topic is brought up in the past decade because of the thrives in many fileds such as Artificial Intellegence. Take Alpha Go as an example, if given long enough time for each step of making choice, nothing will be fancy here because the laptop on your hand can find a good choice too. However, the difference here is Alpha go is smart enough to finish computing in just few minutes and gives an awesome result. Thus, we are now at a stage of chasing “speed” as well as accuracy. Inspired by CS61C, I’ll introduce a method called parallel programing, which will make computing much fatser. In this post, I’ll mainly focus on thread-level parallel computing in R language and take matrix multiply and mutual outlink problems as examples to demonstrate how Optimize Programming takes the advantage of using more potentials of our computers

What is thread level parallisim

A simple analogy will make understanding this much easier. Suppose we have to move 8 bags from Place A to Place B, yet if we only have one workers, we need to move 8 times, but if we have 8 workers, we may finish them just in 1 time!

Backgroud of library parallel

The parallel library is now a build in library for R and it can be called directly by library(parallel). Another useful package which we should download is the “snow” library, how we use that will be demonstrated later in the post

```
library(parallel)
library(snow)
```

```
##
## Attaching package: 'snow'
```

```
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, clusterSplit, makeCluster,
##     parApply, parCapply, parLapply, parRapply, parSapply,
##     splitIndices, stopCluster
```

Preparation

First we’ll talk about the function detectCores() which take in a void and return the number of physical cores your computer has. By doing this we actually knows how many threads(how many workers) R can ask the Operating system to parallel with, now as demonstrated below, my computer has 8 cores and now I’ll assume my computer can run 8 threads at the same time for simplicity (for curious, my CPU is intel 7 which support a technic called “hyper-threading”" which may run more than expected).

```
cores <- detectCores()
cores
```

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

And another function System.time, which is the timing function in r. It takes in a function you want to timing and return a table of timing result. The following is a simple adding problem which is timed by R.

```
system.time({
  num = 0
  for (i in 1:100){
    num = num + i
  }
})
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

Matrix Multilply

Now let’s start the naive multiply, which is the warm-up for the mutual outlinks problem, the idea is letting everyone know how matrix multiply works and think about how we can parallel in matrix operating before we step further. The following codes is a naive way multiply our matrix, I’ll not apply any tricks here. The function Matrix_set_up takes in size of row and returns a row * row array filled up with 1 and another matrix filed up with 2. It will return a result matrix wich is computed by m1 multiply m2.

```r
# Matrix_set_up gives out a random matrix with given size
Matrix_set_up <- function(size_row)
{
  # First matrix set up
  m1 <- matrix(sample(1,size_row * size_row,replace=T), nrow = size_row)
  # Second matrix set up
  m2 <- matrix(sample(2,size_row * size_row,replace=T), nrow = size_row)
  # result sets up
  result <- matrix(sample(0,size_row * size_row,replace=T), nrow = size_row)
  # iterate the row of m1
  for (i in 1:size_row) {
    #iterate the column of m2
    for (j in 1:size_row) {
      #iterate inside each row/column
      for (k in 1:size_row) {
        #[i,j] = returns
        result[i,j] = m1[i,k] * m2[k,j]
      }
    }
  }
  return(result)
}

# time our 100 * 100 matrix multiply
system.time({Matrix_set_up(100)})
```

```
##    user  system elapsed
##   0.104   0.000   0.104
```

Mutual Outlinks Problem

Now we get some flavors in how to do matrix operation in R and probably you will get some ideas on how we split our works for each threads, now consider the following Pseudocode for function mtl (in below chunks):

```
    sum = 0;
    for i:0,1...,n-1
      for j:i+1....,n-1
        for k:0,1,....n-1
          sum += a[i,k] * a[j,k]
```

The code above has three iterations which is similar to our matrix multiplication before, and the first iteration goes through the number of row and the second go through the number of column, the last iteration goes into each row or column. The difference here is : for the second iteration, we actually start from the i, instead of 0.
The work spliting seems clear now, we can split each j for i since they are independent, or we can also split i directly, the only thing we can't do is spliting k because this will give an error (we can't make sure threads finish sequentially, that's to say if thread 1 update sum but thread 2 is also running and it has sum = 0, so without knowing sum is updated by thread 1, this will give us a wrong result) the following shows how spliting i works:

```
    sum = 0;
      when i = 0  ->  thread 1
      for j:i+1....,n-1
        for k:0,1,....n-1
          sum += a[i,k] * a[j,k]

      when i = 1  ->  thread 2
      for j:i+1....,n-1
        for k:0,1,....n-1
          sum += a[i,k] * a[j,k]
      ......

      when i = n ....
```

Below is the mtl function.

```r
mtl <- function(ichunk,m){
  n <- ncol(m)
  matches <- 0
  for (i in ichunk) {
    if (i < n) {
      rowi <- m[i,]
      matches <- matches + sum(m[(i+1):n,] %*% rowi)
    }
  }
  matches
}
```

This part is the core on spliting works, step into the code chunks and see my comments.

```r
# Driver function for the multlink Problem. It takes in a cluster, and an array that we want to apply mtl function on. By cluster it's actually threads over here, which means number of parts you want to split out to work in the same time.
mutlinks <- function(cls,m) {
  # rows in matrix m
  n <- nrow(m)
  # num of threads we work on
  nc <- length(cls)
  # split the work with the split function
  options(warn = -1)
  ichunks <- split(1:n,1:nc)
  options(warn=0)
  #apply the function with all threads
  counts<- clusterApply(cls, ichunks,mtl,m)
  # find the mean = sum / (n*(n-1)/2)
  do.call(sum,counts)/(n*(n-1)/2)
}
```

Below is runing with one thread and we measure the times.

```r
system.time(
{
# create clusters, this case we only have one clusters, "localhost" means the thread on you computer(number of workers).
cl <- makeCluster(type="SOCK", c("localhost"))
# initialize a 1000 * 1000 matrix filled with 0 or 1.
testm <- matrix(sample(0:1,1000*1000,replace=T),nrow=1000)
# call driver functions above.
mutlinks(cl,testm)
# when we end computing, return the resource to computer.
stopCluster(cl)}
)
```

```
##    user  system elapsed 
##   0.062   0.007   7.344
```

Below is runing with two thread and we measure the time.

```r
system.time(
{
# Now we combine two local host and it gives me 2 thread(workers)
cl <- makeCluster(type="SOCK", c("localhost","localhost"))
testm <- matrix(sample(0:1,1000*1000,replace=T),nrow=1000)
mutlinks(cl,testm)
stopCluster(cl)}
)
```

```
##    user  system elapsed 
##   0.110   0.009   4.181
```

Cool, isn't it? we actually achieves a speed up by around 2 times.

A Problem That We Shouldn't Ignore

Now it comes up a question, is it always good if we can prallel as much as we can ? Below is runing with four thread and we measure the time.

```r
library(snow)
system.time(
{cl <- makeCluster(type="SOCK", c("localhost","localhost","localhost","localhost"))
testm <- matrix(sample(0:1,1000*1000,replace=T),nrow=1000)
mutlinks(cl,testm)
stopCluster(cl)}
)
```

```
##    user  system elapsed 
##   0.048   0.015   3.067
```

Below is runing with four thread and we measure the time.

```r
library(snow)
system.time(
{cl <- makeCluster(type="SOCK", c("localhost","localhost","localhost","localhost",
                                  "localhost","localhost","localhost","localhost"))
testm <- matrix(sample(0:1,1000*1000,replace=T),nrow=1000)
mutlinks(cl,testm)
stopCluster(cl)}
)
```

```
##    user  system elapsed
##   0.091   0.028   3.858
```

As we see that the speed actually goes down from 4 to 8, and parallel 4 threads don't gives us 4 times faster as expected. Back to our analogy at begining, though you have 8 workers, but now consider the work as "finish one paper cutting"…. So do we really need 8 workers to finish papper cutting? Probably some workers have nothing to do and they are actually talking to each other which even slow down the progress!

Conclusion And A Quick Outlook

Above all, we have seen how to improve our matrix multiply in R, and indeed, we see a real speedup by using parallel computing; however, this is just a basic trick on using parallel computing, and as we seen that what we are supposed to get 4 times or 8 times faster are not actually achieved. It is due to the following: First of all, as I mentioned before, using 1-d Array may perform better. Second, we don't use the full potential of our computer memory hierachy. Lastly, the R language itself is restricted by how it compiles and that's the barrier what we want to look into, by using a more "hardware-related" language, we may achieve a better performance, which is indeed a tremendous improvment. The way we can do this is importing the C code in R and run OpenMp(or even CUDA for GPU Accelerate) technique on them.

Take-Home Message

1. system.time() is the timing function you need to get and detectCores() help you know the total number of physical cores your computer has.
2. The idea of parallel computing is "split your works".
3. Thread level computing can be acheived by R snow library function makeCluster(), which allocates workers for you, remember to call stopCluster() so that you won't run into warnings.
4. Performance is actually restricted and the speed improvment is not linear.
5. R is actually not the best language to apply parallel computings.

Reference

1. "The ART OF R PROGRAMMING" by Norman Matloff ISBN:978-1-59327-384-2
2. "Professional CUDA C Programming" by John Cheng, Max Grossman, Ty McKercher ISBN:978-1-118-73932-7
3. "How-to go parallel in R – basics + tips" from web page "http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/"
4. "Cookbook for R - measuring elapsed time"" from web page "http://www.cookbook-r.com/Scripts_and_functions/Measuring_elapsed_time/"
5. "Simple Network of Workstations for R" from web page "http://homepage.divms.uiowa.edu/~luke/R/cluster/cluster.html"
6. "Cluster Analysis" from web page "https://www.statmethods.net/advstats/cluster.html"
7. "Split Document" from web page "https://www.rdocumentation.org/packages/base/versions/3.4.1/topics/split"