Parallel_R

Cheng Ju

3/16/2015

Useful reference

- For more information about the foreach package in R, you can visit the web: http://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf
- For more information about the arguments in doParallel, you can visit the web: http://cran.r-project.org/web/packages/doParallel/doParallel.pdf
- and for more examples, I suggest you to read http://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf
- Most parts are cited from workshop by Chris Paciorek in stat department. Full workshop will cover more materials, which you can find in his webpage: http://www.stat.berkeley.edu/~paciorek/

Note:

to allow for parallelization, before starting the BCE VM, go to Machine > Settings, select System and increase the number of processors. You may also want to increase the amount of memory.

Basic shared memory parallel programming

A simple way to exploit parallelism in R when you have an embarrassingly parallel problem (one where you can split the problem up into independent chunks) is to use the foreach package to do a for loop in parallel. For example, bootstrapping, random forests, simulation studies, cross- validation and many other statistical methods can be handled in this way. You would not want to use foreach if the iterations were not independent of each other.

The doParallel package is a "parallel backend" for the foreach package. It provides a mechanism needed to execute foreach loops in parallel. The foreach package must be used in conjunction with a package such as doParallel in order to execute code in parallel. The user must register a parallel backend to use, otherwise foreach will execute tasks sequentially, even when the %dopar% operator is used.

Getting started

```r
library(doParallel)
```

```
## Loading required package: foreach
## Loading required package: iterators
## Loading required package: parallel
```

```r
x <- foreach(i=1:4) %do% sqrt(i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
##
## [[4]]
## [1] 2
```

This is a bit odd looking, because it looks vaguely like a for loop, but is implemented using a binary operator, called %do%. Also, unlike a for loop, it returns a value. This is quite important. The purpose of this statement is to compute the list of results. Generally, foreach with %do% is used to execute an R expression repeatedly, and return the results in some data structure or object, which is a list by default.

So far, all of our examples have returned a list of results. This is a good default, since a list can contain any R object. But sometimes we'd like the results to be returned in a numeric vector, for example. This can be done by using the .combine option to foreach:

```r
x <- foreach(i=1:4, .combine='c') %do% exp(i)
x
```

```
## [1]  2.718  7.389 20.086 54.598
```

```r
y <- foreach(i=1:4, .combine='cbind') %do% rnorm(5)
y
```

```
##      result.1 result.2 result.3 result.4
## [1,]  -1.3221   0.8270   0.8471  0.17838
## [2,]  -0.2151  -0.6871   0.3980  0.68368
## [3,]   0.7097   1.3656   0.7984 -1.25002
## [4,]  -0.3500   1.5467   1.3428  0.57045
## [5,]   0.1999  -0.1470  -2.0217  0.08819
```

You can also specify a user-written function to combine the results.

Apply functions to array

The Collatz conjecture is a conjecture in mathematics named after Lothar Collatz, who first proposed it in 1937.

Take any natural number n. If n is even, divide it by 2 to get n / 2. If n is odd, multiply it by 3 and add 1 to obtain 3n + 1. Repeat the process indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1. The property has also been called oneness

```r
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)

func <- function(x) {
      n = 1
      raw <- x
      while (x > 1) {
            x <- ifelse(x%%2==0,x/2,3*x+1)
            n = n + 1
      }
      return(c(raw,n))
}
```

```r
size<-10000
Eptime<- system.time({
      psteps <- foreach(x=1:size,.combine='rbind') %dopar% func(x)
      stopCluster(cl)
})
## Return which number will take most iteration to get 1.
psteps[which.max(psteps[,2]),]
```

```
## [1] 6171  262
```

We can compare the running time with sequential programming. Here we can also use foreach. But we use %do% rather than %dopar%.

```r
Estime<- system.time({
      ssteps <- foreach(x=1:size,.combine='rbind') %do% func(x)
})
ssteps[which.max(ssteps[,2]),]
```

```
## [1] 6171  262
```

```r
Eptime
```

```
##    user  system elapsed
##   3.908   0.310   6.283
```

```r
Estime
```

```
##    user  system elapsed
##   5.561   0.016   5.580
```

Exercise

If we want to calculate the variance of some statistic, we can use boostrap. For example, we can use boostrap to calculate the variance of median of exponential distribution. How to do this only use function "foreach", "median", and "sample"?

Statistical model based on parallel

Random forest is easy to do by parallel programming. For argument “combine” of foreach, we can directly use “combine” so collect the results in different cores.

Here is an example using doParallel and foreach package to do randorm forest.

```
library(doParallel)
library(randomForest)
```

```
## randomForest 4.6-10
## Type rfNews() to see new features/changes/bug fixes.
```

```
prf.time<-system.time({
      cl <- makeCluster(4)
      registerDoParallel(cl)
      ## each cluster, we set 2500 trees
      rf <- foreach(ntree=rep(2500, 4),
                    .combine=combine,
                    .packages='randomForest') %dopar%
            randomForest(Species~., data=iris, ntree=ntree)
      stopCluster(cl)
})
## directly use sequential programming getting 10000 trees
srf.time<-system.time(
      randomForest(Species~., data=iris, ntree=10000)
)

prf.time
```

```
##    user  system elapsed
##   0.890   0.058   2.062
```

```
srf.time
```

```
##    user  system elapsed
##   0.791   0.051   0.849
```

Here is an example using doParallel and foreach package to do GLM based on bootstraped data.

```
library(doParallel)

#making cluster. Here we set the number of clusters equal to 2.
cl <- makeCluster(2)
registerDoParallel(cl)


## iris dataset
x<- iris[which(iris[,5]!='setosa'),c(1,5)]
trials<- 100
# Parallel Computing
ptime<- system.time({
      ## icount count number of times that the iterator will fire.
      r1<- foreach(icount(trials), .combine=cbind) %dopar% {
            ## Do sample from index 1 to 100. Sample 100 times with repalcement.
            ind<- sample(100,100,replace=T)
            ## Here we use logistic model. We use glm function used sampled data
            result1<- glm(x[ind,2]~x[ind,1],family=binomial(logit))
            coefficients(result1)
      }
})
## Compared with Sequatial Computing
stime<- system.time({
      r2<- foreach(icount(trials), .combine=cbind) %do% {
            ind<- sample(100,100,replace=T)
            result1<- glm(x[ind,2]~x[ind,1],family=binomial(logit))
            coefficients(result1)
      }
})

ptime
```

```
##    user  system elapsed
##   0.045   0.004   0.264
```

```
stime
```

```
##    user  system elapsed
##   0.295   0.007   0.301
```

Exercise

- How to use the model to do prediction using parallel programming?

As you may find from some resules. If the model is too simple, like random forest and bootstraping, it may not be a good idea to use parallel programming, because it will take too much time communication with each core compared with sequential programming. However, when using more complex model like GLM, we can save a lot of time by using parallel programming.

So far we have discussed some samples of shared memory parallel programming. Sometimes we may need parallel in distributed memory. If you can do your computation on the cores of a single node using shared memory, that will be faster than using the same number of cores (or even somewhat more cores) across multiple nodes. Similarly, jobs with a lot of data/high memory requirements that one might think of as requiring Hadoop may in some cases be much faster if you can find a single machine with a lot of memory.

How to use SCF/biostat cluster

You can find some details here: http://statistics.berkeley.edu/computing/servers/cluster#access-job-restrictions

How to upload files to cluster

- Before connect to cluster, change into your working direction

- In command line, type: **sftp user@arwen.berkeley.edu**
- upload files (for example, you want use simulate.R): **put simulate.R**
- close sftp: **quit**

You can also use "get" command to get files from cluster.

In this step, we are not using the shell of remote computer. Hence "tap" can not be used. You should make sure the name of file is correct.

How to run code in cluster

- Prepare a **shell script** containing the instructions you would like the system to execute. For example a simple script to run an R program called '*simulate.R*' would contain the single line: **R CMD BATCH –no-save simulate.R simulate.out**

- Connect to cluster: **ssh user@arwen.berkeley.edu**

- In Unix-like operating systems, chmod is the command and system call which may change the access permissions to file system objects (You can think it like a permission for the file): **chmod 755 job.sh**

- You shoule see "Your job XXXXXXXX ("job.sh") has been submitted". If you want to see status of your job, just type in command line: **qstat**

- After you type 'qstat', you will just look at 'queue', you will find something like this: *low.q@scf-sm001.Berkeley.EDU*. This is the mathine that running your code. If you want find details, just use **qrsh -q interactive.q -l hostname=scf-sm001** and **top** (you can use control+c to exit from top) command to find detail information. After that, you can type exit to quit.

- Finally, just look at your out file: **cat Simulate.Rout**