Parallel Programming in R

Kejia Hu

1. Review the Idea of Parallel Programming

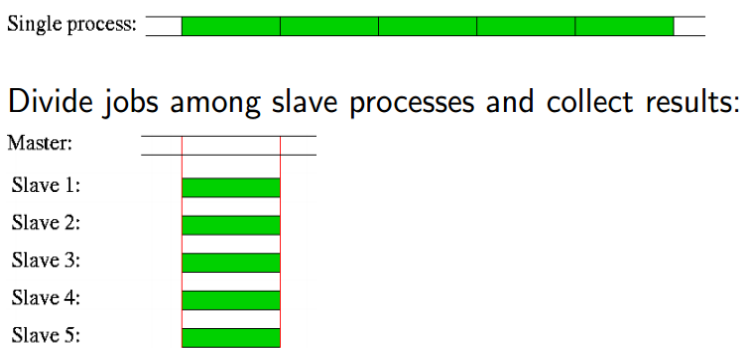
Parallel computing is widely used in high-performance computing. The idea is to break large problems into smaller ones that can be solved concurrently or called "in parallel". Using the master-slave model as an example, the ideal performance improvement should be p processors should be p times faster than one processor. In this sense, the 1-minute work with single processor should only cost 2 seconds to complete with 30 processors. While the original 1-year work with single processor can be finished in two weeks. The gain in saving time is very impressive and thus stimulates us to adopt parallel programming in large data problems.

Before we fill the page with parallel programming jargon, we fist introduce some basic terminology.

- A **CPU** is the main processing unit in a system. Sometimes CPU refers to the processor, sometimes it refers to a core on a processor, it depends on the author.
- A **cluster** is a set of machines that all interconnected and share resources as if they were one giant computer
- The **master** is the sytem that controls the cluster
- A **slave** is a machine that performs computations and responds to the master's requests.

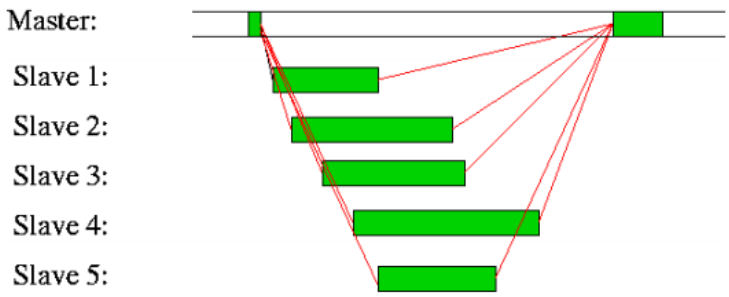
Under the ideal case, we illustrate the idea of parallel programming in Figure 1. The computation speed will be p times faster with p slaves.

Figure 1: Master/ Slave Model-Ideal Case



However, in reality as illustrate in Figure 2, the jobs for slaves are very likely to vary in complexity and machines' computation power is different. Also the communication between master and slaves take up some time in dividing up jobs and collecting the result. Thus it is not guarantee that the speed will be p times faster with p slaves.

Figure 2: Master/Slave Model-Realistic Case



In order to evaluate and compare the packages available in R for parallel programming, we grasp an overview of the existing technologies for parallel programming. In general, parallel computing deals with hardware and software for computation in which many calculations are carried out simultaneously. The main goal of parallel computing is the improvement in calculating capacity.

- Hardware: Multi-processor and multi-core computers have multiple processing elements which share main memory together within a single machine.
- Software:
  - Shared memory: Shared memory programming languages communicate by manipulating shared memory variables. The most common ones are OpenMP and Pthreads. . Recent work indicates that is easier to program with OpenMP than Pthreads and that OpenMP delivers faster execution times (Breshears and Luong 2000; Binstock 2008).
  - Distributed memory: Message-passing APIs are widely used in distributed memory systems. The master-slave model is most common. Well-known implementations include PVM and MPI. A comparison of MPI and PVM approaches is available in Gropp and Lusk (2002). For computer clusters PVM is still widely used, but MPI appears to be emerging as a de-facto standard for parallel computing.

2.Compare Packages

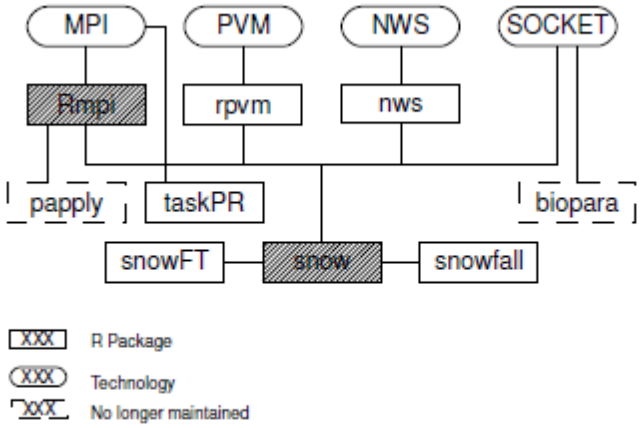
Several different R packages are available for cluster-based parallel computing. In the following context, we will compare some of the widely-used ones from three perspectives: system level, code level and speed level.

2.1. System level

R packages for computer cluster are rmpi, rpvm, nws, snowfall, snow, snowFT, taskPR and papply. In this report, I will only explore several of them.

Package	Technology	Launch R instances
Rmpi	MPI	At the slaves from the master and close until it is asked
rpvm	PVM	Directly execute R script at slaves and close after running
snow	Rmpi, rpvm, nws, socket	At the slaves from the master and close until it is asked

For the technology support, PVM is widely used but MPI is the best supported communication protocol for parallel computing with R. Higher-level package snow can then deploy Rmpi.
Figure 1 shows the family tree of available R packages with the support technology. The plot is adopted from paper "State of the Art in Parallel Computing with R".



2.2. Code level

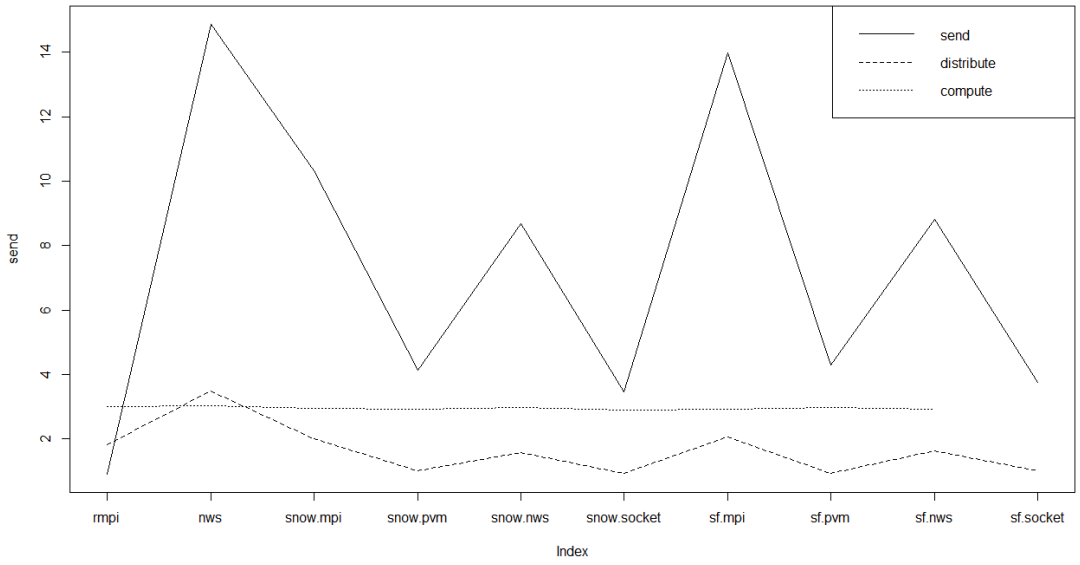
The usability table refers to the paper "State of the Art in Parallel Computing with R". It describes the difficulty level for certain software to achieve a particular goal. Each package is evaluated from the following aspects. Learnability: how easy for first-time users to learn functionality. Efficiency: how quickly users can perform tasks. Memorability: how easily users can reestablish details required for use. Errors: how to avoid errors and to fix them. Satisfaction: how pleasant to use a package. The assessment range from ++ to --.

	Learnability	Efficiency	Memorability	Errors	Satisfication
RmpiR	+	--	++	+	+
Rpvm	--	--	+	-	--
Nws	+	+	++	+	+
Snow	+	++	++	+	++
snowFT	+	++	++	+	++
snowfall	++	++	++	+	++
Papply	+	+	++	+	0
taskPR	+	-	++	0	-

Overall, Rmpi and rpvm are complex since user must interact with communication API at a low level. The packages rpvm provide very basic functions for parallel computing while package snow, nws and Rmpi have scripts to launch R instances at the workers. The packages snowFT and snowfall use scripts from the package snow.

2.3. Speed level

Here we compare the speed of packages in the following test: sending data from the manager to all workers, distributing a list of data from the manager to the workers and classic parallel computation example. The experimental code is from http://www.jstatsoft.org/v31/i01/supp/1.



From the plot, we see that the most time consuming part is sending data from the manager to all workers, then the parallel computation part and last is the distribution work. Also, from the above plot, for both distribution and computation, snow packages with socket consume the least time while nws consumes the most. In general, socket connection is the least time consuming for sending, distribution and computation compared with nws and mpi. Moreover, the advantage of snow package and snowfall package start to show in distribution and computation.

3. Application of Parallel programming

Parallel programming can be applied in statistical computation such as cross-validation and bootstrap.

3.1. Example: Bootstrap

```
#single cluster: 1000 replicates on 1 cluster
    wallTime(nuke.boot <-boot(nuke.data, nuke.fun, R=1000, m=1,
                              fit.pred=new.fit, x.pred=new.data))
#Parallel version: 100 replicates on each of 10 cluster nodes:
```

```
wallTime(cl.nuke.boot <-clusterCall(cl,boot,nuke.data, nuke.fun, R=100, m=1,
                                    fit.pred=new.fit, x.pred=new.data))
```

Using parallel computing substantially improve the speed of bootstrap.

3.2.Example: Cross Validation

```
#using for loop
    for(i in 1:d){
       fitlm <-lm(speed~.,cars[-ind[[i]],])
       res[i]<-sum(fitm$reside^2)
    }

#using sequential mapping function
    res2<-unlist(lapply(1:d,function(i){
      fitlm<-lm(speed~.,cars[-ind[[i]],])
      sum(fitlm$resid^2)}))

#parallel function
    cl<-makeCluster(4,"MPI")
    res4<-unlist(parLapply(cl,1:d,function(i,ind){
      fitlm<-lm(speed~.,cars[-ind[[i]],])
      sum(fitlm$resid^2)
    },ind))
   stopCluster(cl)
```

In different experiments, we see the the speed of parallel comunting the fastest and then it is the sequential mapping function and then the loop function.

Appendix_ R code

```
library(snow)
#Create a cluster of 10 R slave processes:
cl <- makeCluster(10)
#Find out where the processes are running:
do.call("rbind", clusterCall(cl, function(cl) Sys.info()["nodename"]))
#Stop the cluster
stopCluster(cl)
#Call function on all nodes:
clusterCall(cl, exp, 1)
#Evaluate an expression on all nodes:
clusterEvalQ(cl, library(boot))
#Apply function to list, one element per node:
clusterApply(cl, 1:5, get("+"), 2)
#Parallel lapply
unlist(parLapply(cl, 1:15, get("+"), 2))
#Parallel sapply
parSapply(cl, 1:15, get("+"), 2)
#Parallel apply
parApply(cl, matrix(1:10, ncol=2), 2, sum)
#Random number generation needs help:
clusterCall(cl, runif, 3)
#Identical streams are likely, not guaranteed.
library(rsprng)
clusterSetupSPRNG(cl)
clusterCall(cl, runif, 3)
#Example: Parallel Bootstrap
#1000 replicates on a single processor:
wallTime <- function(expr) system.time(expr)
# In this example we show the use of boot in a prediction from
# regression based on the nuclear data.    This example is taken
# from Example 6.8 of Davison and Hinkley (1997).    Notice also
# that two extra arguments to 'statistic' are passed through boot.
nuke <- nuclear[, c(1, 2, 5, 7, 8, 10, 11)]
nuke.lm <- glm(log(cost) ~ date+log(cap)+ne+ct+log(cum.n)+pt, data = nuke)
nuke.diag <- glm.diag(nuke.lm)
nuke.res <- nuke.diag$res * nuke.diag$sd
nuke.res <- nuke.res - mean(nuke.res)

# We set up a new data frame with the data, the standardized
# residuals and the fitted values for use in the bootstrap.
nuke.data <- data.frame(nuke, resid = nuke.res, fit = fitted(nuke.lm))
```

```
    nuke.fun <- function(dat, inds, i.pred, fit.pred, x.pred)
    {
      lm.b <- glm(fit+resid[inds] ~ date+log(cap)+ne+ct+log(cum.n)+pt,
                  data = dat)
      pred.b <- predict(lm.b, x.pred)
      c(coef(lm.b), pred.b - (fit.pred + dat$resid[i.pred]))
    }
    # Now we want a prediction of plant number 32 but at date 73.00
    new.data <- data.frame(cost = 1, date = 73.00, cap = 886, ne = 0,
                           ct = 0, cum.n = 11, pt = 1)
    new.fit <- predict(nuke.lm, new.data)

    wallTime(nuke.boot <-boot(nuke.data, nuke.fun, R=1000, m=1,
                              fit.pred=new.fit, x.pred=new.data))
    #Parallel version: 100 replicates on each of 10 cluster nodes:
    wallTime(cl.nuke.boot <-clusterCall(cl,boot,nuke.data, nuke.fun, R=100, m=1,
                                        fit.pred=new.fit, x.pred=new.data))
    #Example: Cross Validation
    #using for loop
    for(i in 1:d){
      fitlm <-lm(speed~.,cars[-ind[[i]],])
      res[i]<-sum(fitm$reside^2)
    }

    #using sequential mapping function
    res2<-unlist(lapply(1:d,function(i){
      fitlm<-lm(speed~.,cars[-ind[[i]],])
      sum(fitlm$resid^2)}))

    #parallel function
    cl<-makeCluster(4,"MPI")
    res4<-unlist(parLapply(cl,1:d,function(i,ind){
      fitlm<-lm(speed~.,cars[-ind[[i]],])
      sum(fitlm$resid^2)
    },ind))
stopCluster(cl)
```