UNIVERSITY OF TEXAS AT SAN ANTONIO

Parallel Computing with R and How to Use it on High Performance Computing Cluster

Liang Jing

Nov. 2010

1 Abstract

Methodological advances have led to much more computationally demand in Statistical computing, such as Markov chain Monte Carlo (MCMC) algorithms, bootstrapping, cross-validation, Monte Carlo simulation, etc. And many areas of statistical applications are experiencing rapid growth in the size of data sets. Parallel computing is a common approach to resolve these problems. In this paper, four most promising packages for parallel computing with R are introduced with examples of implementation. And the procedure about how to use R in parallel way on high performance computing cluster is illustrated.

2 Introduction

Multi-processor and multi-core computers, either in the form of personal computer (PC) or high performance computing (HPC) cluster, have become much more accessible nowadays. So it is desired and necessary to optimize the calculation with the help of parallel computing techniques.

Though R, as one of the most successful languages in the past ten years, has countless cheerful features, it suffers two major limitations: first, R reads data into memory by default which limits the size of data to read and could exhaust RAM by storing unnecessary data easily; second, R will only utilize one processor under the default build regardless of the number of available processors. However, there come a variety of packages developed for the rescue:

- {Rmpi} by Yu 2002
- {snow} by Rossini et al. 2007
- {snowfall} by Knaus 2008
- {multicore} by Urbanek 2009
- more: {rpvm}, {papply}, {taskPR}, ...

In this paper, the first four packages are introduced shortly and illustrated with sample codes about their usage. Then their implementation on high

performance computing (HPC) cluster is explained as well as the job management system on HPC. More details about parallel computing with R can be found in Schmidberger et al. 2009.

3 PACKAGES

3.1 RMPI

Rmpi is a wrapper to *message-passing interface* (MPI) which is a standardized message-passing system designed to function on a wide variety of parallel computers. Rmpi provides a R interface on top of low-level MPI functions so that R users don't need to worry about details of MPI implementations. The package includes scripts to launch R instances on the slaves from the master and data and functions will be sent to slaves. Then the master will communicate with the slaves to perform the computation and collect results. Besides, it also provides several R-specific functions, such as `mpi.parApply()`, and some error handling mechanism to report errors from the slaves to the master.

3.2 SNOW

snow is short for *Simple Network of Workstations*. It provides an interface to several parallelization packages: MPI via {Rmpi}, *NetWork Spaces* (NWS) via {nws}, *Parallel Virtual Machine* (PVM) via {rpvm}, and raw *Sockets* via the operating system. This means it is possible to run the same code at a cluster or PC with PVM, MPI or NWS.

To use this package, we need to start with creating a number of "workers" of the specified type, "SOCK", "PVM", "MPI", or "NWS" (details of these types are not introduced here).

```
cl <- makeCluster(4, "MPI")
```

Then there are two types of functions in the package can be used for parallel computing: intermediate level functions and high level functions.

INTERMEDIATE LEVEL FUNCTIONS

`clusterCall(cl, fun, x)` : asks each worker process to call function "fun" with argument "x" and returns a vector of the results.

```
print( clusterCall(cl, function()
Sys.info()[c("nodename","machine")])
)
clusterCall(cl, min, rnorm(5))
clusterCall(cl, function(x) x, 1:2)
```

`clusterApply(cl, vec, fun)` : assigns one element of the vector "vec" to each worker, has each worker call the function "fun" on its assigned element, and collects the result. If there are more elements than workers then workers are reused cyclically.

```
print( clusterApply(cl, 1:5, function(x)
Sys.info()[c("nodename","machine")])
 )
clusterApply(cl, 1:10, function(x) print(1:x))
```

A load balancing version of `clusterApply(cl, vec, fun)` is

```
clusterApplyLB(cl, 1:10, function(x) print(1:x))
```

HIGH LEVEL FUNCTIONS

`parLapply(), parApply(), ...` These higher level functions are easy to implement in terms of the intermediate level functions, for example `clusterApply()`:

```
parLapply <- function(cl, x, fun, ...) {
  y <- splitList(x, length(cl))
  val <- clusterApply(cl, y, lapply, fun, ...)
  docall(c, val)
}
```

Here `splitList(x, length(cl))` uses `splitList()` to split the input into P approximately equal-sized pieces for P worker processes.

```
splitList(1:10, 4)
```

A SIMPLE EXAMPLE

A common computing problem is to apply a specified function on each row of the data. Here let's simply apply `mean()` on each row of a big matrix. The serial (nonparallel) way of this computation will be

```
n <- 1e5
s <- matrix(rnorm(n*100),n,)
# 1) serial way
v <- apply(s, 1, mean)
```

To use intermediate level functions in {snow}, we need to divide data, then run each chuck on different workers.

```
# 2) parallel way by using clusterApply()
cl <- makeCluster(4, "MPI")
idx <- clusterSplit(cl, 1:dim(s)[1])
ssplt <- lapply(idx, function(i) s[i,])
v2 <- clusterApply(cl, ssplt, function(x) apply(x, 1, mean) )
stopCluster(cl)
```

And using high level functions is just as simple as using common `apply`-like functions. Sometimes, it could be slower than using intermediate level function and require more memory.

```
# 3) parallel way by using parApply()
cl <- makeCluster(4, "MPI")
v3 <- parApply(cl, s, 1, mean)
stopCluster(cl)
```

A CROSS-VALIDATION EXAMPLE

Here is the core part of codes for cross-validation by using `for` loop.

```
# Step 1: write code draft by using for loop
for(i in 1:d){
fitlm <- lm(speed~., cars[-ind[[i]],])
res[i] <- sum(fitlm$resid^2)
}
```

Aiming to use high level `apply`-like functions in {snow}, we need to convert the code of `for` loop into a sequential mapping function such as `lapply()`.

```
# Step 2: rewrite code in terms of a sequential mapping function
# such as lapply().
res2 <- unlist(lapply(1:d, function(i){
  fitlm <- lm(speed~., cars[-ind[[i]],])
  sum(fitlm$resid^2)
  }
))
```

Then replace `lapply()` with its parallel version high level function `parLapply()`.

```
# Step 3: turn the code into a parallel function by adding a
# cluster argument and replacing the call to lapply by a call
# to parLapply()
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)
```

3.3 SNOWFALL

snowfall is a top-level wrapper to snow by adding an abstraction layer for better usability. snowfall API is very similar to snow API and has the following features:

- functions for loading packages and sources into clusters;
- functions for exchanging objects between clusters;
- changing cluster setting can be done from command line and does not require changing R code;
- all wrapper functions contain extended error handling;

- all functions also work in sequential mode and R code can run in both modes without modification;
- it is designed as a connector to the cluster management tool `sfCluster` but can work without it.

Another wonderful feature of snowfall is *load balancing* which enables faster clusters for further work instead of waiting for slower clusters when they finish current work. Also, snowfall provides handy `apply`-like variants: `sfLapply()`, `sfSapply()`, `sfApply()`, `sfRapply()`, `sfCapply()`.

To use {snowfall}, we start with generating a number of "workers" similar to using {snow}.

```
# Initialisation of snowfall.
# (if used with sfCluster, just call sfInit())
sfInit(parallel=TRUE, cpus=4, type="SOCK")
```

Then data sets and packages that are required for computation need to be exported to each worker.

```
# Exporting needed data and loading required
# packages on workers.
sfExport("sir.adm")
sfLibrary(cmprsk)
```

And random number generators need to be set up for each worker.

```
# Start network random number generator
# (as "sample" is using random numbers)
sfClusterSetupRNG()
```

For the computation desired to be parallelized, write a wrapper function to encapsulate the code. Here is an example of fitting randomly selected data with linear regression.

```
# Wrapper function, which can be parallelized.
wrapper <- function(idx) {
# Output progress in worker logfile
# cat( "Current index: ", idx, "\n" )
index <- sample(1:nrow(sir.adm), replace=TRUE)
```

```
temp <- sir.adm[index, ]
fit <- lm(temp$time~ temp$status+ temp$pneu)
return(fit$coef)
}
```

Then use appropriate `apply`-like parallel function to perform the computation of wrapper function. And don't forget to stop the cluster.

```
result <- sfLapply(1:run, wrapper)
sfStop()
```

3.4 MULTICORE

While previous three packages can be categorized as explicit parallelism where users control most of cluster settings, multicore package offers implicit parallelism where the messy work in setting up the system and distributing data is abstracted away. In multicore, all jobs share the full state of R when parallel child instances are spawned and spawning uses the "fork" system call or operating system specific equivalent. This mechanism results in fast spawning and the advantage that no data or code needs to be copied or initialized on child processes, but also limits its usage on Windows system because Windows lacks the "fork" system call (experimental support for Windows is available though).

Three major functions in multicore offer two ways to do parallel computation. First, `mclapply()` is a parallel version of `lapply()` with extra parameters that control parallel settings. Second, `parallel()` and `collect()` functions allow to create child processes, perform parallel computation and retrieve the results in a more manual fashion. In addition, there are low-level functions available but usually not recommended to use, such as `fork()`, `process()` and `sendMaster()`.

When using `mclapply()`, we can set up the number of processes to distribute the job in two ways: either set up the computational environment outside;

```
options(cores=4) # number of cores
getOption("cores")
```

or set the parameter inside

```
mclapply(1:30, rnorm, mc.cores = 4, mc.set.seed = FALSE)
```

where the parameter `mc.set.seed` is used to decide whether use different seeds for different processes.

To use `parallel()` and `collect()` mechanism, the suggested way is:

1) write a wrapper function that contains the procedure to be parallelized, for example some big matrix computation

```
wrapper <- function(x){
  A <- matrix(rnorm(1e6),1e3,)
  det(solve(A)%*%A) }
```

2) use `lapply()` to start a number of parallel jobs that will run the wrapper function

```
jobs <- lapply(1:10, function(k) parallel(wrapper(k)) )
```

3) perform running and retrieve the results from all the processes

```
collect(jobs)
```

When executing `collect(jobs)`, the jobs scheduled by `parallel()` will spawn simultaneously and all available processors will be occupied to run these jobs until all the jobs are done.

4 HPC CLUSTER

A typical HPC cluster usually consists of hundreds or thousands of processors, for example "cheetah.cbi.utsa.edu" cluster at UTSA has 392 processing cores with 2 GB RAM per core. Considering it serves for many individual clients to perform different computing jobs, a job management system (also called job scheduler) is installed on the cluster to handle the job requests from all the clients, schedule and assign available processors to perform the jobs. To run our program in parallel way on cluster, we need to write a job script that describes our request of computing resource (how many processors we need? for how long?) and computing environment.

For Sun Grid Engine system installed on "cheetah.cbi.utsa.edu", a sample job script is shown below.

```
#!/bin/bash
# The name of the job
#$ -N R-parallel
# Giving the name of the output log file
#$ -o R_parallel.log
# Combining output/error messages into one file
#$ -j y
# One needs to tell the queue system
# to use the current directory as the working directory
# Or else the script may fail as it will execute
# in your top level home directory /home/username
#$ -cwd
# Here we tell the queue that we want the orte parallel
# enivironment and request 5 slots
# This option take the following form: -pe nameOfEnv min-Max
#$ -pe orte 5-10

# Now come the commands to be executed

setenv PATH ${PATH}:/share/apps/gsl/bin
setenv LD_LIBRARY_PATH /share/apps/gsl/lib:${LD_LIBRARY_PATH}

/opt/openmpi/bin/mpirun -n $NSLOTS R --vanilla
< test_snow_bootstrap.R > test_snow_bootstrap.log

exit 0
```

After the job parameters are set up, the path to the libraries required for the computation needs to be added into environment and then the R code can be executed in batch mode.

Another way to utilize many processors on HPC cluster is submitting job arrays which will be executed on different processors individually. A sample job script is shown below.

```
#!/bin/bash
#$ -N R-array
#$ -o R_jobarray.log
```

```
#$ -j y
#$ -cwd
#$ -t 1-10
# -M liang.jing@utsa.edu
# -m e

setenv PATH ${PATH}:/share/apps/gsl/bin
setenv LD_LIBRARY_PATH /share/apps/gsl/lib:${LD_LIBRARY_PATH}

R --vanilla --args $SGE_TASK_ID < test_jobarray.R
```

where `-t 1-10` describes how many times the program will be repeated with the task ID `1-10` assigned to each job. By taking the parameter `$SGE_TASK_ID` as an input argument into R, the R program will be able to perform different codes for each running job. The corresponding R code should be in the following form.

```
### test_jobarray.R
myarg <- commandArgs()
id <- as.numeric(myarg[length(myarg)])

if(id == 1) {
...
}
if(id == 2) {
...
}
...
```

More details about how to write job scripts can be found on Oxford e-Research Centre (`http://www.oerc.ox.ac.uk/computing-resources/osc/support/documentation-help/job-schedulers/`).