Big Data Computing Cases with SupR

Yongqiang Lian

The China-R Conference (10th, Shanghai)

East China Normal University

December 03, 2017

Agenda

1. Introduction
2. Get Started
3. Multithreading
4. Cluster Computing

1 Introduction

- SupR was developed by professor Chuanhai Liu in Department of Statistics, Purdue University, and it is still in development. SupR was made possible by modifying R (R-3.1.1) existing internal system implementation with additional 40K lines of new source code in C. The R syntax is however kept unchanged.
- The Java-like multithreading for parallel computing.
- The Spark-like built-in SupR functions for distributed computing.
- R is functional language, and everything in R is object.

2 Get Started

2.1 Computer operating systems

- Linux Ubuntu

2.2 Download and install

- While there is still much to do, a private pre-release is available at http://www.stat.purdue.edu/~chuanhai/SupR/release/
- Restore the archive:

```
$ cd ~/Downloads
$ cd /
$ sudo tar -xvf ~/Downloads/SupR-1.0.0-ubuntu-14.04.tar
```

2.3 Define environment variables

- With the bash command shell, add environment variable `$SUPR_HOME` to `$HOME/.bashrc`.

```
export SUPR_HOME='/opt/SupR-1.0.0'
```

- Optional environment variables can also be defined.

```
export SUPR_MASTER_HOST='localhost'
export SUPR_MASTER_PORT=5118
export SUPR_GLOBAL_DIR=$HOME'/.SupR'
```

- Start a SupR session in a command window.

```
$ $SUPR_HOME/bin/R
```

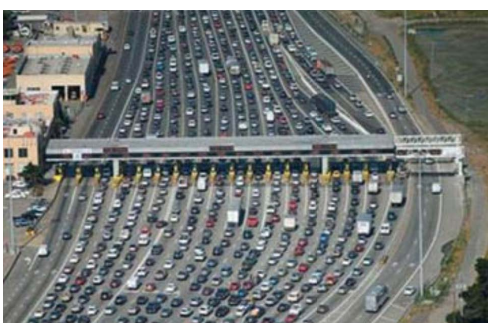
3 Multithreading

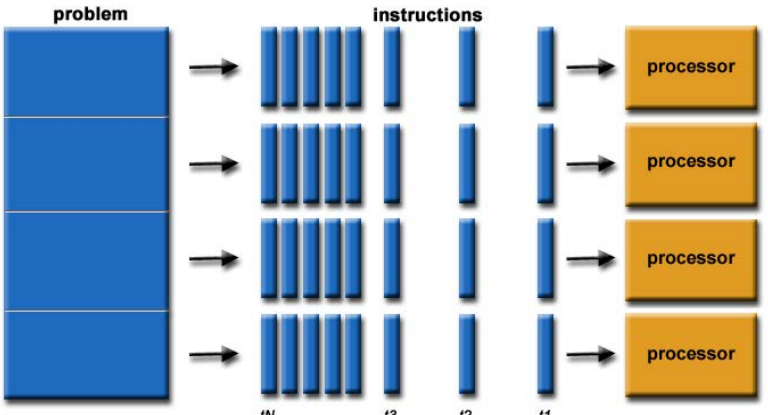
- Every object is a function. An internal function, named `implicit`, is included for convenience to associate functions with non-function objects.

```
N <- 100(1000)(1000)
sec <- 1
minute <- 60(sec)
```

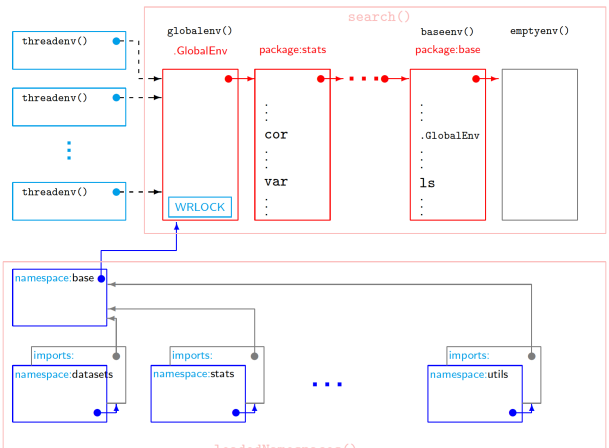
```
5(minute) == 300(sec)
A <- matrix(1:10, 2, 5)
A(t(A))
```

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.




problem
instructions
processor
processor
processor
processor
tN
t3
t2
t1

- SupR thread name spaces



- SupR thread functions:

```
new.thread()
start.thread()
join.thread()
current.thread()
cancel.thread()
thread.sleep()
interrupt()
sync.eval()
wait()
notify()
......
```

```
new.thread(1 + 2, start = TRUE)
```

- SupR graphics

```
# start a X11 graphics.server
new.thread(X11())
current.thread()
plot(1:10000000, rnorm(10000000), type = "l",
     main = current.thread())
rnorm(100)
alive.threads()
```

3.1 Parallel EM via SupR multithreading

- Start a SupR session and define two simple functions to simulate and split an incomplete normal sample.

```
simu <- function(N = 1000(1000), mis.frac = 0.25, mu = 8.0){
  X <- rnorm(N) + mu
  M <- runif(N) < mis.frac
  X[M] <- NA
```

```
  X
}
split <- function(X, np){
  N <- length(X)
  ncols <- np
  nrows <- as.integer((N-1)/ncols) + 1
  matrix(c(X, rep(NA, nrows*ncols-N)), nrows, ncols, byrow = T)
}
```

- Create simulated subsets.

```
data <- simu(mis.frac = 0.5)
NP <- 16
subsets <- data.frame(split(data, np = NP))
```

- Define the likelihood function.

```
log.likelihood <- function(){
  z <- data[!is.na(data)] - MU
  -sum(z*z)/2
```

```
}
```

- Define colors to plot the likelihood sequence, and start X11 graphics running in a separate thread.

```
max_iter <- 500L
Colors <- rainbow(2*max_iter)[1:max_iter]
new.thread(X11())
```

- Specify the starting values of the EM algorithm.

```
MU <- 0
SS <- list(sum.x = 0, n = 0, n.threads = NP)
sync.m <- "another object"
sync.e <- "one more object"
```

- Create and start a new thread to run the M-step of EM.

```
M.thread <- new.thread(
  env = list2env(list(max_iter = max_iter, NP = NP)),
  {
```

```
cache(TRUE)
nocache(SS, MU)
sync.eval(sync.m, {
log.likelihood <<- log.likelihood
environment(log.likelihood) <- environment()
ll <- numeric(0)
iter <- numeric(0)
for(i in 1:max_iter){
  cat("\n\033[0;32mM-step is ready ...\033[0m\n")
  wait(sync.m)
  MU <<- SS$sum.x/SS$n
  SS <<- if(i == max_iter) NULL else list(sum.x = 0, n = 0, n.threads = NP)
  sync.eval(sync.e, notify(sync.e, all = TRUE))
  ll <- c(ll, log.likelihood())
  iter <- c(iter, i)
  cat(current.thread(), "\033[0;34mM-Step: iteration", i, MU, "\033[0m\n")
  plot(iter, ll, pch = 16, col = Colors[iter], xlab = "iteration",
       ylab = "log.likelihood")
```

```
    }})
  }, start = TRUE)
```

- Create and start NP new threads to run the E-step of EM.

```
E.threads <- as.list(1:NP)
for(i in 1:length(E.threads)) E.threads[[i]] <- new.thread(
  env = list2env(list(X = subsets[[i]])),
  {
    cache(TRUE)
    nocache(SS, MU)
    mis <- is.na(X)
    while(!is.null(SS)){
      X[mis] <- MU
      ss <- list(sum.x = sum(X), n = length(X))
      sync.eval(sync.e, {
              SS <<- list(sum.x = SS$sum.x + ss$sum.x, n = SS$n + ss$n,
                          n.threads = SS$n.threads - 1)
              cat("\033[0;33m", current.thread(), "\tn.threads =", SS$n.threads,
```

```
                "\033[0m\n")
            if(SS$n.threads == 0) sync.eval(sync.m, notify(sync.m))
            sync.eval(sync.e, wait(sync.e))
        })
    }
  }
)
for(i in 1:length(E.threads)) start.thread(E.threads[[i]])
```

4 Cluster Computing

- A SupR cluster consists of
  1. one master session
  2. a number of worker sessions
  3. one driver session
- The SupR cluster has its own simple built-in distributed file system, defined

dynamically by running worker sessions. The service is provided and managed by the built-in block managers and a block manager master.

- To start a master session, start a SupR session and call the `start.master` function.

```
start.master()
```

- To start a worker session, start a SupR session and call the `start.worker` function.

```
start.worker()
```

- To start a driver session, start a SupR session and call the `start.driver` function.

```
start.driver()
```

4.1 Gaussian linear regression model

- Define the two functions in the driver session.

```
# simulate data from Y = X*beta + sigma*rnorm(m)
# returns an iterator that produces n subsample of size m
lr.subsets <- function(n, m, p, beta, sigma = 1){
  XY <- NULL
  has.next <- function(){
    if(!is.null(XY)) return(TRUE)
    if(n == 0) return(FALSE)
    n <<- n - 1
    X <- matrix(c(rep(1, m), rnorm(m*p)), nrow = m)
    Y <- X(beta) + sigma(rnorm(m))
    XY <<- cbind(X, Y)
    return(TRUE)
  }
  get.next <- function(){
    if(is.null(XY)) stop("cannot find the next value")
    xy <- XY
```

```
    XY <<- NULL
    return(xy)
  }
  return(iterator(has.next, get.next))
}

# the sweep operator
sweep <- function(S, K){
  det <- attr(S, "det")
  ln.det <- if(is.null(det)) 0.0 else log(det)
  for(k in K){
    flag <- sign(k)
    k <- abs(k)
    a <- S[k,k]
    ln.det <- ln.det + log(flag*a)
    S[k,k] <- -1/a
    S[-k,-k] <- S[-k,-k] - S[-k,k,drop=F]%*%S[k,-k,drop=F]/a
    a <- a*flag
```

```
    S[k,-k] <- S[k,-k]/a
    S[-k,k] <- S[-k,k]/a
  }
  structure(S, det = exp(ln.det))
}
```

- Simulate and distribute by evaluating the following expressions in the driver session.

```
p <- 4
beta <- rnorm(1+p)
sigma <- 0.1
iter <- lr.subsets(100, 10(1000), p = p, beta, sigma = sigma)
distribute(iter, name = "lr", np = 1, nrep = 2)
```

- Call `map.reduce` to obtain the class of the distributed subsets.

```
map.reduce(lr, function(x) class(x[[1]]), c)
```

- Call `map.reduce` to compute the sufficient statistics.

```
ss <- map.reduce(lr, function(x) t(XY <- x[[1]])(XY), `+`)
```

- Do LS estimation with the sufficient statistics using the sweep operator.

```
N <- as.integer(ss[1,1])
P <- dim(ss)[1] - 1
H <- sweep(ss, 1:P)
beta.hat <- H[P+1,-(P+1)]
sigma.hat <- sqrt(H[P+1,P+1]/(N-P))
data.frame(beta.hat, beta)
c(sigma.hat, sigma)
```

Thank You!