BigR: Multithreaded and Distributed R

1. BigR Multithreading

Chuanhai Liu

Department of Statistics, Purdue University

February 8, 2016
What is R?

- R is R.
  
  *To turn ideas into software, quickly and faithfully.*

- R is functional language, and everything in R is object.
- There are a lot of good things that can be said about R.
  ...

- But as far as I know, R doesn’t do multithreading, a desirable feature for parallel and distributed computing for big data analysis.
The BigR experiment of modifying/rewriting R for BigData analysis was started with R-3.1.1.

The R language syntax is not changed.

Let’s start a BigR session ...

To warn up a bit and for fun, let’s make every object a function.

```r
> N = 100(1000)(1000)
> sec = 1
> minute = 60(sec)
> 5(minute) == 300(sec)
```
An internal function, named `implicit`, is included for convenience to associate functions with non-function objects. The R-interface is given by

```r
function (class, fun = NULL) {
  if (missing(class))
    class = ""
  .Internal(implicit(as.character(class), fun, NULL))
}
```

The `implicit` function associates the function object, provided by the `fun` argument, with object of class `class`. For `class = ""`, it returns the list of current associations. For any non-function object `instance` of class `class`, the function call `instance(...)` is equivalent to `fun(instance, ...)`. 

```r
> implicit()
> A = matrix(1:10, 2, 5)
> A(t(A))
```
Toward BigData analysis, a more serious one is to introduce a pair of functions to form what is called an iterator object.

```r
> iter = as.iterator(1:10)
> while(has.next(iter))
    print(get.next(iter))
```

# A convenient tool to make a non-function object into a "function" object
```r
> implicit("iterator", function(x, FUN){
    while(has.next(x))
        FUN(get.next(x))
})
```

```r
> iter = as.iterator(LETTERS)
> iter(print)
```
The `has.next` takes an iterator object and returns a logical value, indicating the availability of the next element. When `has.next` returns TRUE, the next element will be provided by calling `get.next` with the same iterator object.

```
iterator = function(has.next, get.next, env=parent.frame()){
  .Internal(iterator(has.next, get.next, env))
}

as.iterator = function(x, env=parent.frame()){
  .Internal(as.iterator(x, env, NULL))
}

has.next = function(iter, ...){
  .Internal(has.next(iter, pairlist(...)))
}

get.next = function(iter, ...){
  if(nargs()==1) .Internal(get.next(iter, list(...))) else
    .Internal(get.next(iter, pairlist(...)))
}
```

Note: These R-interface functions are subject to refinement for the first release of BigR.
It is very helpful to use the R X11 graphics for data analysis.

The BigR X11 graphics is simply the R graphics, with a separate thread managing redrawing of X11 windows.

Discuss an example here ...
# Simulate a normal sample with missing values

data = simu(N = 10(1000)(1000), mis.frac=0.5)
NP = 4; subsets = data.frame(split(data, np = NP))

# The problem is to find the MLE of mu, MU, using EM
MU = 0

# Complete-data Sufficient Statistics
SS = list(sum.x=0, n=0, n.threads = NP)

# Use two objects for concurrency
sync.m = "another object"
sync.e = "one more object"

Note: The simu and split functions can be found on the last slide.
M.thread = new.thread(
    env = as.environment(list(max_iter = 100, NP=NP)),
    sync.eval(sync.m, {
    for(i in 1:max_iter){
        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))

        cat(current.thread(),"M-Step: iteration", i, MU, "\n")
    }
}), start=TRUE)
Threaded EM Algorithm: E Threads

```r
e.threads = as.list(1:NP)

for(i in 1:length(e.threads))
  e.threads[[i]] = new.thread(
    env=as.environment(list(x=subsets[[i]])),
    {
      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$n.threads - 1)
            if(SS$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(e.threads[[i]])
```
Multithreading: “Ten” basic functions

1. `new.thread`
2. `start.thread`
3. `current.thread`
4. `cancel.thread`
5. `interrupt`
6. `sync.eval`
7. `wait`
8. `notify`
9. `set.synchronized`
10. `is.synchronized`

More on Synchronization, in addition to set.synchronized

- Each new thread has its (local) environment with the global environment, `.GlobalEnv` or `globalenv()`, as its parent environment.
- `.GlobalEnv` is protected with a read-write lock object.
- By default, all packages, installed as parents of `.GlobalEnv`, are read only.
The simu and split functions in the EM example

simu = function(N = 10(1000), mis.frac = 0.25, mu=8.0)
{
    X = rnorm(N) + mu
    M = runif(N) < mis.frac
    # the observed data, from which the EM algorithm finds the MLE of mu
    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)
}