REVOLUTION ANALYTICS

Parallel Tools in R

Revolution Analytics

Outline

1 Parallel Tools in R

2 foreach

3 iterators

4 Nested foreach

5 Exercises

Overview

- Introduction to `foreach()`
- Iterators
- Using `doRSR`
- Using Other Parallel Backends

Parallelizing Loops

rxExec() May be all you need if you and the people you share code with are all using Revolution R Enterprise

foreach() Fully open-sourced, available with any version of R, works with a wide variety of parallelization platforms

General form:

```
library(foreach)
foreach(range) %do% {
  ...
}
## OR ##
foreach(range) %dopar% {
  ...
}
```

Simple example:

```
library(foreach)
foreach(i = 4:6) %do% sqrt(i)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 2.236068
##
## [[3]]
## [1] 2.44949
```

Functionally equivalent to:

```
lapply(4:6, sqrt)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 2.236068
##
## [[3]]
## [1] 2.44949
```

More about `foreach()`

- The `%do%` operator runs `foreach` expressions sequentially
- The `%dopar%` operator attempts to run them parallely if a suitable parallel backend is registered.
- If no parallel backend is registered, `%dopar%` acts like `%do%` and runs the expression sequentially.
- `foreach()` is not the fastest way to run parallel programs, BUT it is extremely portable.
- You can write a parallel program using `foreach()` and run it with a wide variety of backends, so it is almost always possible to run it in parallel, no matter the configuration of computer or cluster.

A Plethora of Parallel Back Ends

- `doRSR`, `doParallel`, `doSNOW`, `doMC`– what do they all mean?
- `doRSR` and `doSNOW`
  - Cluster oriented
  - Work on both multi-CPU* and clusters
  - Allow `foreach()` to work with RevoScaleR and snow packages, respectively
- `doParallel` (R 2.14.0 and later)
  - combines doSNOW/doMC as the associated parallel package combines snow and multicore

A Plethora of Parallel Back Ends

- `doMC`
  - allows `foreach()` to work with multicore package, only available on Linux (needs fork, which is not available, as such, on Windows)
- For completeness, `doSNOW` / `doParallel` can also work with parallel packages `Rmpi` and `rpvm`
  - Interfaces to MPI and PVM, respectively
  - `Rmpi` might be valuable for large clusters for performance reasons
- A `doNWS` package is available to work with `NetWorkSpaces`.

Use of Back Ends is Similar

- Load library for parallel backend
  - `library(doRSR|doSNOW|doParallel|doMC)`
- Create a set of workers
  - RSR: use `RxHpcServer()` or `RxLsfCluster()` to create a cluster compute context and register compute context
  - `snow` or `parallel`: `cl <-makeCluster(<nodes>);`
  - `MC`: none needed
- Register the back end
  - registerDoRSR()
  - registerDoSNOW(cl) or registerDoParallel(cl)
  - registerDoMC()

Outline

1. Parallel Tools in R
2. foreach
3. iterators
4. Nested foreach
5. Exercises

Simple Example Using `foreach()`

Compute list of square roots

```
## Use package parallel
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)
foreach(i = 1:4) %dopar% sqrt(i)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
...
```

```
stopCluster(cl)
```

Notes/Observations

`sqrt()` is a toy example — it will NEVER be faster in parallel than the built-in sqrt function running on your local system. It shows that `foreach()` and your parallel backend are working correctly, but it is not worth spending any further time profiling.

Note: Use package `snow` if package `parallel` is not available; steps are essentially the same (replace `doParallel` with `doSNOW`)

A More Serious Example: Simulation

- The game of craps is a simple casino game involving a pair of dice.
  - If you roll a 7 or 11 on your first roll, you win
  - if you roll a 2, 3, or 12 you lose
  - If you roll 4, 5, 6, 8, 9 or 10, that number becomes your point.
  - You continue rolling until you roll either a 7 (in which case you lose) or your point (in which case you win).

Naive craps function in R

```
playCraps <- function() {
  result <- NULL;  point <- NULL
  count <- 1
  while (is.null(result)) {
    roll <- sum(sample(6, 2, replace=TRUE))
    if (is.null(point)) { point <- roll }
    if (count == 1 && (roll == 7 || roll == 11)) { result <- "Win" }
    else if (count == 1 && (roll == 2 || roll == 3 || roll == 12)) { result <- "Loss" }
    else if (count > 1 && roll == 7 ) { result <- "Loss" }
    else if (count > 1 && point == roll) { result <- "Win" }
    else { count <- count + 1 }
    result
    }
  result
  }
```

Using `foreach()` to Run the Simulation

```
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)
ngames <- 10000
z1 <- foreach(i = 1:ngames) %dopar% playCraps()
table(unlist(z1))
```

```
##
## Loss  Win
## 5115 4885
```

Exercise

Think of the SNOW and mandelbrot examples.

Why is `playCraps()` naive?

How can you make it more efficient?

Can you also adjust it to simulate the distribution of the number of rolls for wins vs. losses?

Outline

1 Parallel Tools in R

2 foreach

3 iterators

4 Nested foreach

5 Exercises

iterators

- Iterators are a generalization of looping variables—they are objects that define a sequence, but do not contain the sequence.
- Two parts to iterators
  1. Create an iteration object: `iterobj <- iter( 1:10 )`
  2. Access the object: `nextElem( iterobj )`
- Works with:
  - sequences
  - matrices
  - data frames
  - Functions

Some iterator Functions

Iterator	Functionality
`iter()`	Creates an iterator
`irnorm(<n>)`	Uterator to generate values of normal distribution
`irunif()`	Iterator for uniform distribution
`irbinom()`	Iterator for binomial distribution
`irnbinom()`	Iterator for negative binomial distribution
`irpois()`	Iterator for poison distribution
`ircount(<n>)`	Equivalent to iter`(1:3)`

Ex: irnorm(4, count = 1000) Creates 1000 sets of 4 random numbers each

Notes:

- iterator creates data as needed, not all at beginning
- Can improve performance especially in parallel execution

Matrix Example using iter()

```
library(iterators)
matobj <- iter(matrix(1:4, ncol = 2))
a <- matrix(1:4, ncol = 2)
nextElem(matobj)
```

```
##      [,1]
## [1,]    1
## [2,]    2
```

- Notice that we did not create a matrix
- Iterators can increase parallelism (remember Amdahl's law)

Combining iter with foreach

```
library(doParallel)
cl <- makeCluster(4)
registerDoParallel(cl)
dim <- 1000
matobj <- iter((matrix(rnorm(dim^2), ncol = dim)), by = "col")
foreach(i = matobj, .combine = cbind) %dopar% mean(i)
```

```
##         result.1    result.2   result.3    result.4     result.5
## [1,] -0.01592259 -0.02619414 0.05910607 0.005091903 -0.004085502
##        result.6     result.7   result.8    result.9  result.10 result.11
## [1,] 0.01456528 0.0004798503 0.03068613 -0.04822302 0.05831266 -0.029898
##         result.12   result.13  result.14    result.15  result.16
## [1,] -0.009012115 -0.01664641 0.04665384 -0.005928236 0.03370931
##         result.17   result.18    result.19   result.20  result.21
...
```

Combining iter with foreach

```
a <- matrix(rnorm(dim^2), dim)
foreach(i = 1:dim, .combine = c) %dopar% mean(a[, i])
```

```
##    [1] -0.00253163072 -0.00468538827 -0.00751721752 -0.04534430828
##    [5] -0.00005126006 -0.00012133651 -0.02773000873 -0.03930346686
##    [9]  0.03634293508 -0.01844192804  0.01974050014  0.01506064384
##   [13] -0.00909810400 -0.05723182219 -0.01000236607  0.01170986935
##   [17]  0.04628218120 -0.04693123307 -0.02876159038 -0.01607473776
##   [21] -0.02073334598  0.04573983282 -0.00965400040 -0.06221283337
##   [25] -0.02182951712  0.01089553127 -0.02095173663  0.02826171968
...
```

```
stopCluster(cl)
```

Benefits

Two things happen here where we use an iterator.

- First, the host doesn't create the matrix, which saves some time. Whatever that time is, it is divided between the number of workers.
- Second, each worker creates its own data via the iterator. Neither worker needs to wait for the master to send it its portion of the matrix. An important sequential part of the program has been removed.

Outline

1 Parallel Tools in R

2 foreach

3 iterators

4 Nested foreach

5 Exercises

Nested for loop

We'll use the vignette example as is:

```
sim <- function(a, b) 10 * a + b
avec <- 1:2
bvec <- 1:4
x <- matrix(0, length(avec), length(bvec))
# use for loop first:
for (j in 1:length(bvec)) {
  for (i in 1:length(avec)) {
    x[i, j] <- sim(avec[i], bvec[j])
  }
}
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   12   13   14
## [2,]   21   22   23   24
```

And Now with Nested foreach

```
x <- foreach(b = bvec, .combine = "cbind") %:% foreach(a = avec, .combine = "c") %do%
  {
    sim(a, b)
  }
```

Note:

- %:% acts to combine the inner and outer loops
- Should not think of this as nested for loops but …
- As merged loop structure
- Easily parallelized, but where to put %dopar% ?

Parallelizing Nested foreach

- Parallelizing outer loop is usually most efficient
- However, nested `foreach()` does not create an inner and outer loop
- Creates stream of operations easing parallelization
- Simply replace `%do%` with `%dopar%`

%do% to %dopar%

```
x <- foreach(b = bvec, .combine = "cbind") %:% foreach(a = avec, .combine = "c") %do%
  {
    sim(a, b)
  }
```

Becomes:

```
x <- foreach(b = bvec, .combine = "cbind") %:% foreach(a = avec, .combine = "c") %dopar%
  {
    sim(a, b)
  }
```

- Simply changing %do% to %dopar% parallelizes this nested `foreach()` loop
- Need to be sure you have a parallel backend registered (and that it is using a valid cluster!)

Parallelization Summary

- RevoScaleR solves many R and parallel R issues
  - Large memory usage not an issue
  - Low-cost parallelization
  - Important set of statistical functions built-in (e.g. summary statistics, linear regression, binomial logistic regression, cross tabs)
- The `RevoScaleR` function `rxExec()` lets you tackle “the rest” of HPC
- The function `foreach()` with `%dopar%` give a portable solution

`.RProfile` – Making Life Easier

- Sets start-up options for your programs, e.g.
  - Load the same packages for every program
  - Always use the same node list
  - Always use the same launch mechanism
- R looks in current and home directories for `.Rprofile`
  - Where is that?
  - use `getwd()` to find

When Things Goes Wrong (Windows)

- R creates processes
  - All may not shut down
- Parallel backend may create additional processes.
- What to do?
  - Shut down R Enterprise for Windows
  - End all `Rterm.exe` processes (`Rterm.exe` can stay around – can't move files used by them)

Outline

1 Parallel Tools in R

2 foreach

3 iterators

4 Nested foreach

5 Exercises

Lab

- The data frame `mtcars` has 32 observations of 11 variables. We are interested in the effect of displacement (`disp`) on gas mileage (`mpg`).
- Use `foreach()` to perform a simple bootstrap regression on this dataset.
  - Detailed accounts of the bootstrap are available in Davison and Hinkley (1997) and Efron and Tibshirani (1993). A brief introduction to bootstrap regresssion can be found online at `http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-bootstrapping.pdf`

Steps to Solution

- Perform simple regression
- Bootstrap the regression (i.e. sample observations with replacement, and rerun the regression)
- Combine the results

Going further

- Try recasting this example as an `rxExec()` call.
- Hint: Take the braced expression on the right hand side of `%dopar%` and turn it into a function.

Session Summary

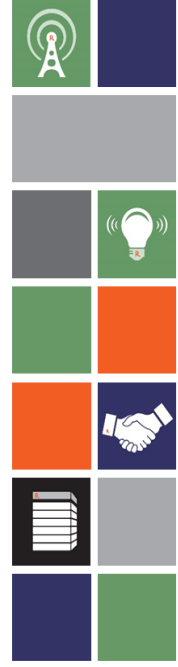
- More parallel tools
  - `foreach()` with `%dopar%`
  - iterators
  - nested `foreach()`
- Creating clusters
- package `parallel`
  - Makes parallelizing apply-based programs simple

References

foreach User Guide http://packages.revolutionanalytics.com/doc/7.3.0/linux/RevoForeachIterators_Users_Guide.pdf

Questions?

REVOLUTION ANALYTICS



Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com
1.855.GET.REVO
Twitter: @RevolutionR