Lecture 17: Parallel Processing in R and Rcpp

STAT 385 - James Balamuta

July 28, 2016

On the Agenda

- Administrative Issues
  - HW5 Due on August 7th
  - Group Presentations on August 2nd during class
- Parallel in *R*
  - `doParallel`
- Parallel with *Rcpp*
  - `OpenMP`
- Extra: Implicit Parallelism

`foreach` and `do*`

- RevolutionAnalytics has created very nice wrapper functions that enable the conversion of R loop code to a parallelized version.
- Specifically, the `foreach` package implements parallelized looping.
- Packages that have the prefix `do` correspond to a parallelization backend. (e.g. `doParallel`)
- **Using the parallel backend in turn makes loops faster!**

*under certain assumptions.

`foreach` and `do*` - Assumptions

Faster loops come with a couple assumptions:

- There is no dependency within the loop.
- The calculation is not trivial.

Otherwise, the overhead associated with the parallelizing the loops will destroy any gain and possible make results slower to obtain.

foreach

The syntax for a `foreach()` loop is slightly different than a `for()` loop:

```
# Foreach
foreach(i = 1:3) %do% { expression }

# Base R for loop
for(i in 1:3){ expression }
```

Note:

1. The `% %` after the `foreach` declaration and before the `{ }` is a binary operator.
2. `i = 1:3` vs. `i in 1:3`

`foreach` binary operators

- The `foreach` loop requires a binary operator be placed after it.
- The binary operators are:

Operator	Description
`%do%`	Sequential foreach loop (e.g. R's old loop)
`%dopar%`	Parallel foreach loop
`%dorng%`	Obtain reproducible randomization (requires `doRNG` package)
`%:%`	Indicates a nested loop

Sample `foreach` loop

Here is a simple foreach loop. Note that the output is that of a list.

```
library("foreach")
foreach(i = 1:3) %do% { sqrt(i) }
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
```

foreach Output Types

- By modifying .combine="", the output shifts from being a list to being a vector.
- To store the results from the foreach loop, assign a variable to it!

```
x = foreach(i = 1:3, .combine='c') %do% { sqrt(i) }

x
```

```
## [1] 1.000000 1.414214 1.732051
```

`foreach` Output Types

Common `.combine=""` types:

- `c` for concatenating the results into a vector
- `cbind` for concatenating the results into a column-based matrix
- `rbind` for concatenating the results into a row-based matrix

Warning!

This loop structure will NOT recycle values! Make sure your index and additional params are of the similar length

```
x = foreach(a = 1:50, b = c(1,2), .combine='c') %do% {
  a + b
}

x
```

```
## [1] 2 4
```

Using Packages in Parallel

- In particular, with parallel code comes the necessity to load packages on all open threads.
- To do so, use: `.packages=c('package_name')`

```
# Loading packages on different workers
foreach(i = 1:3, .packages=c('gmwm')) %dopar% {
  expression
}
```

foreach setup - parallelization

```
require(doParallel, quiet = TRUE) # Load library

cl = makeCluster(2)     # Create cluster for snow

registerDoParallel(cl) # Register it with doParallel

getDoParWorkers()       # See how many workers exist
```

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

```
# Do a parallelized loop!
foreach(i = 1:3, .combine='c') %dopar% { sqrt(i) }
```

```
## [1] 1.000000 1.414214 1.732051
```

```
stopCluster(cl)         # End cluster for snow
```

Nested `foreach`

- Often, there needs to be more than one loop active at a given time.
- Thus, “nesting” or placing one loop within another.

Nested `foreach`

Below shows nesting in a sequential case with for and foreach loops.

```
# Standard R loop
out = character()
for(i in 1:3) {
  for(j in c("A","B")) {
    out = c(out,paste0(i,j))
  }
}
```

```
# Nested foreach loop
foreach(i = 1:3, .combine = c) %do% {
  foreach(j = c("A","B"), .combine = c) %do% {
    paste0(i, j)
  }
}
```

```
## [1] "1A" "1B" "2A" "2B" "3A" "3B"
```

Nested `foreach`

With foreach loops, you can only execute one loop out of the nested conditions in parallel. Thus, it is ideal if you set the "overloop" or the outer loop to be parallel.

```
cl = makeCluster(2)     # Create cluster for snow
registerDoParallel(cl) # Register cluster with doParallel

# foreach with overloop parallelized
foreach (i = 1:3, .combine = c,
         .packages = 'foreach') %dopar% {

  foreach(j = c("A","B"), .combine = c) %do% {
    paste0(i, j)
  }

}
```

```
## [1] "1A" "1B" "2A" "2B" "3A" "3B"
```

Nested foreach

```
cl = makeCluster(2)     # Create cluster for snow

registerDoParallel(cl) # Register cluster with doParallel

# foreach with parallel using %:%
foreach (i = 1:3, .combine = c) %:%
  foreach(j = c("A","B"), .combine = c) %dopar% {
    paste0(i, j)
  }
```

```
## [1] "1A" "1B" "2A" "2B" "3A" "3B"
```

```
stopCluster(cl)         # End cluster for snow
```

Reproducible Parallelization

- If the objective of your work is to make sure the results are reproducible, then it is important to set seeds.
- However, setting a seed within the parallel code is not ideal since it assumes the same job will be distributed to each core each time.
- **This is not true and a dangerous assumption**
- The solution? Establish a set of seeds ahead of time and pass them along with the job.

Reproducible `foreach`

To create a reproducible `foreach` loop, we need to

1. Set a seed with `registerDoRNG()`.
2. Use %dorng% from `doRNG` package as the binary operator.

```
library(doRNG)          # Load doRNG
```

```
## Loading required package: rngtools
## Loading required package: pkgmaker
## Loading required package: registry
##
## Attaching package: 'pkgmaker'
## The following object is masked from 'package:base':
##
##     isNamespaceLoaded
```

```
cl = makeCluster(2)     # Create cluster for snow
```

`foreach` Summary

- Make sure to register your backend via `registerDoParallel(cl)`
- The binary operator `%dopar%` or `%dorng%` must be used for a parallel `foreach`.
- `%do%`, `%dopar%`, or `%dorng%` must appear on the same line as the `foreach()` call.

Application of Parallelization: Bootstrapping

- When we compute a statistic on the data, we only know that one statistic.
- As a result, we don't see how variable that statistic may be.
- To solve this conundrum, Efron proposed Bootstrapping in the now famous 1979 paper entitled: "Bootstrap methods: another look at the jackknife".
- This is a **loop** intensive operation that relies on creating data which "we might have seen."

Application of Parallelization: Bootstrapping

- The Bootstrapping approach necessitated the creation of a large number of datasets from the initial sample by randomly sampling observations with replacement.
- Under this approach, data is generated so that it might of been obtained if we were able to collect data multiple times.
- On each of the data sets, the statistic is then computed leading to the creation of a distribution ontop of the statistic.

A bootstrapping example with `for`...

```
# Get Iris Data
x = iris[which(iris[,5] != "setosa"), c(1,5)]
B = 10000                                  # Iteration
db = matrix(NA, nrow = 2, ncol = B) # Results

system.time({

   for(i in 1:B){                          # Loop
    ind = sample(100, 100, replace=TRUE)
    result = glm(x[ind,2] ~ x[ind,1],
                 family = binomial(logit))
    db[,i] = coefficients(result)
   }

})
```

```
##    user  system elapsed
##  21.174   0.510  23.391
```

A bootstrapping example with `foreach` ...

```
x = iris[which(iris[,5] != "setosa"), c(1,5)]
B = 10000                    # Iteration

system.time({
  db = foreach(icount(B), .combine=cbind) %do% {
    ind = sample(100, 100, replace=TRUE)
    result = glm(x[ind,2] ~ x[ind,1],
                 family = binomial(logit))
    coefficients(result)
  }
})
```

```
##    user  system elapsed
##  25.082   0.493  27.882
```

A bootstrapping example with parallelization...

To make the `foreach` loop parallel we:

1. Change `%do%` to `%dopar%`
2. Add a cluster creation and cluster stop call.

A bootstrapping example with parallelization...

```
x = iris[which(iris[,5] != "setosa"), c(1,5)]
B = 10000                    # Bootstrap iterations

cl = makeCluster(4)          # Create Cluster with 4 cores
registerDoParallel(cl)       # Register cluster for foreach

system.time({
   db = foreach(icount(B), .combine=cbind) %dopar% {
     ind = sample(100, 100, replace=TRUE)
     result = glm(x[ind,2] ~ x[ind,1],
                  family=binomial(logit))
     coefficients(result)
     }
})
```

```
##    user  system elapsed
##   3.052   0.298  13.302
```

Moving along. . .

- That's a wrap on using **foreach**. . . Any questions?
- Moving onto . . . **Parallelizing with Rcpp**

Parallelizing with Rcpp

- Before now, we've discussed how to parallelize components using the R language.
- Sometimes, it is a bit more ideal to use C++ to write algorithms (e.g. lots of loops).
- In this case, we'll talk about using OpenMP with Rcpp or its many variants:
  - RcppArmadillo.
  - RcppEigen.
- Recently, there has been the addition of RcppParallel.
  - But, this addition is more dependent on code functioning within the R ecosystem.

C++ Pragma Directives

The pragma directive is used to access compiler-specific preprocessor extensions.

```
#pragma compiler specific extension
```

- These options are specific for the platform and the compiler you use.
- If the compiler does not support a specific argument for #pragma, then it is ignored with **no syntax error generated**.
- Look at the manual for your compiler for a list of parameters that you can define with #pragma.

Pragma Directive and OpenMP

- Pragma directives are heavily used by OpenMP.
- The directives serve to generate parallelization code.
- Specifically, to parallelize a loop, they are invoked in the form of:

```
unsigned int ncores = 2; // Number of CPUs

#pragma omp parallel num_threads(ncores)
{
  #pragma omp for
  for(i = 0; i < n; i++) { a[i] = a[i] + b[i];}
}
```

Behind the pragma

When using:

```
#pragma omp parallel
```

The pragma directive is converted to:

```
#pragma omp parallel
{
  int this_thread, num_threads, istart, iend;
  id = omp_get_thread_num();
  num_threads = omp_get_num_threads();
  istart = this_thread * N / num_threads;
  iend = (this_thread + 1) * N / num_threads;
  if (this_thread == num_threads-1) { iend = N; }
}
```

So, pragma directives are short cuts or macros for writing code.

Enabling OpenMP for Rcpp

To enable OpenMP support, we can set the required compiler and linker flags within R by:

```
Sys.setenv("PKG_CXXFLAGS"="-fopenmp")
Sys.setenv("PKG_LIBS"="-fopenmp")
```

Or, using Rcpp $\geq$ 0.10.5, we can use a plugin within the C++ code to automatically set these variables for us:

```
// [[Rcpp::plugins(openmp)]]
```

Enabling OpenMP for Rcpp

- The macOS operating environment lacks the ability to parallelize sections of code using the OpenMP standard.
- Typical error:

```
clang: error: unsupported option '-fopenmp'
```

- Read **OpenMP in R on macOS** to enable on macOS.

Protecting OpenMP inclusion for macOS

- However, 99% of the users on macOS, will not have OpenMP enabled compilers.
- As a result, make sure to protect any reference to OpenMP.
- Primarily, protect the inclusion of OpenMP headers with:

```
#ifdef _OPENMP
   #include <omp.h>  // OpenMP header
#endif
```

- Doing so will enable the parallelization of the process on Linux and Windows.
- In the event that Apple enables OpenMP later on, this code will also allow for parallelization to occur.

Sample use of parallelized C++ loop

```
#include <Rcpp.h>
#ifdef _OPENMP
   #include <omp.h>  // OpenMP header
#endif
using namespace Rcpp;
// [[Rcpp::plugins(openmp)]]

// [[Rcpp::export]]
void sample_loop(unsigned int n,
                 unsigned int ncores = 2)
{
  #pragma omp parallel num_threads(ncores)
  {
    #pragma omp for
    for (unsigned int i = 0; i < n; i++){
      // AWESOME_FUNCTION(i);
    }
  }
```

Pragma shortcut

Put the `parallel` and the `for` directive on the same line!

```
#include <Rcpp.h>
#ifdef _OPENMP
   #include <omp.h>  // OpenMP header
#endif
using namespace Rcpp;
// [[Rcpp::plugins(openmp)]]

// [[Rcpp::export]]
void sample_loop(unsigned int n,
                 unsigned int ncores = 2)
{
  #pragma omp parallel for num_threads(ncores)
  for (unsigned int i = 0; i < n; i++){
     // AWESOME_FUNCTION(i);
  }
}
```

Carried Loop Dependency

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector odd_op(NumericVector x){
  unsigned int i, j, n;    // Declare variables
  n = x.size();            // Element size
  j = 1;

  for(i = 0; i < n; i++){
    j += 2;                // Dependency
    x[i] = x[i] + j;
  }

  return x;
}
```

Removing Carried Loop Dependency

- Part of parallelizing requires the loop iterations to be independent.
- Sometimes all that is required is reparameterizing the iteration.

```
// [[Rcpp::export]]
NumericVector odd_op_para(NumericVector x,
                          unsigned int ncores = 2){
  unsigned int i, n;
  n = x.size();

  #pragma omp parallel for num_threads(ncores)
  for(i = 0; i < n; i++){
    unsigned int j = 1 + 2*i; // indep
    x(i) = x(i) + j;
  }

  return x;
}
```

Lowering Dependency: Part II

Question: How do you remove a dependency when you have to combine values into an accumulation variable?

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double average(NumericVector x){
  unsigned int i, n;
  double sum = 0.0;
  n = x.size();

  for (i = 0; i < n; i++) {
    sum += x(i);              // ewk!
  }
  return sum/n;
}
```

Answer: Use a reduction!

Reductions

Reduction operations are the result of needing to combine variables into a single variable. e.g.

```
var = var op expr
```

Requirements for a reduction

1. `var` must be a scalar
2. `expr` a scalar that does not reference var
3. `op` an operation that is either $+$, $*$, or $-$

Pragma directive for reduction:

```
#pragma omp parallel for reduction (op:variable)
```

Reduction in Action

The reduction is able to easily be applied to the prior function we dicussed:

```
// [[Rcpp::export]]
double average_parallel(NumericVector x,
                        unsigned int ncores = 2){
  unsigned int i, n;
  double sum = 0.0;
  n = x.size();

  #pragma omp parallel for reduction(+:sum) num_threads(nco
  for (i = 0; i < n; i++) {
    sum += x[i];
  }
  return sum/n;
}
```

π Example: R to C++

Objective: Make a π approximation function fast.

- Approximation function written in R.

```
pi_me_r = function(){

  num_steps = 100000
  x = sum_x = 0.0

  step = 1.0/num_steps

  for (i in 1:num_steps){
    x = (i + 0.5)*step
    sum_x = sum_x + 4.0/(1.0 + x*x)
  }

  return(step * sum_x) # Pi
}
```

π Example: R to C++

Objective: Make a π approximation function fast.

- Same approximation function written in C++ using Rcpp.

```
#include <Rcpp.h>

// [[Rcpp::export]]
double pi_me(){

  unsigned int num_steps = 100000;
  double x, pi, sum = 0.0;
  double step = 1.0/(double) num_steps;

  for (unsigned int i = 0; i < num_steps; i++){
    x = (i + 0.5)*step;
    sum = sum + 4.0/(1.0 + x*x);
  }
  return step * sum;
}
```

π Example: C++ to Parallelized C++

- Note: The dependency on sum here.
- To parallelize it, we'll need to use a reduction and a private thread.

```
// [[Rcpp::export]]
double pi_me_parallel(unsigned int ncores = 2){
  unsigned int num_steps = 100000;
  unsigned int i; // declared earlier for parallel
  double x, sum = 0.0;
  double step = 1.0/(double) num_steps;

  #pragma omp parallel for reduction(+:sum) num_threads(nco
  for (i = 0; i < num_steps; i++){
    x = (i + 0.5)*step;
    sum = sum + 4.0/(1.0 + x*x);
  }
  return step * sum; // Pi
}
```

π Benchmarks

The speed up of the approximation of π is remarkable!

	test	replications	elapsed	relative	user.self	sys.self
2	cpp.serial	100	0.046	1.000	0.082	0.001
1	cpp.parallel	100	0.074	1.609	0.129	0.001
3	r	100	8.950	194.565	8.190	0.064

Extra Material: Implicit Parallelization

- Next up is Implicit parallelization.
- Please note, that implicit parallelization does not mix well with explicit parallelization due to core allocations needing to be specified in advance.

Base Operations in R

Within R, there are two pieces of software that control how fast a matrix is manipulated and how quickly a numerical routine is run upon it.

- **Basic Linear Algebra Subprograms (BLAS)** performs low-level linear algebra subroutine operations such as copying, scaling, dot products, linear combinations, and matrix multiplication.
- **Linear Algebra Package (LAPACK)** performs numerical routines such as solving linear equations, linear least-squares, and matrix factorizations (SVD, LU, QR, Cholesky and Schur).

Single-Threaded Default BLAS Operations in R

- By default, R ships with a **single-threaded** BLAS and LAPACK.
- This **restricts R** to operating matrix computations on a **single** core.
- For perspective, the Department of Statistics' PCs have either **2 or 4 cores**.
- Thus, there are a considerable amount of computational cycles being wasted.

Parallelizing Base Operations in R

- To parallelize base operations in R, we need to get a new BLAS.
- **The Good News...**
  - BLAS systems normally will ship with an optimized version of LAPACK.
- **The Bad News...**
  - You have to compile R
- **And The Ugly News...**
  - RevolutionAnalytics has its own compiled version of R that uses MKL, but their version of R lags behind developments in R.

Parallelized BLAS Options

- Download and install a new multi-threaded BLAS
  - OpenBLAS: `http://www.openblas.net/`
  - MKL: `https://software.intel.com/en-us/intel-mkl`
  - ATLAS: `http://math-atlas.sourceforge.net/`
- Install guides are only a google away for your specific system.

macOS: Multithreaded vecLib BLAS via symbolic link

macOS users are able to easily switch to a multithreaded BLAS called vecLib by opening Terminal and typing the following:

```
cd /Library/Frameworks/R.framework/Resources/lib

# For R <= 2.15 version (old school R users)
# Line 2:
ln -sf libRblas.vecLib.dylib libRblas.dylib

# For R >= 3.0 version (Latest Install)
# Line 2:
ln -sf /System/Library/Frameworks/Accelerate.framework
/Frameworks/vecLib.framework/Versions/Current/
libBLAS.dylib libRblas.dylib
```

The install differs between the two versions since as of 3.0, R has stopped shipping `vecLib` within the default install.

Benchmarking BLAS change

```
# Create matrix
set.seed(1234)
a = crossprod(matrix(rnorm(3000^2),nrow=3000,ncol=3000))

#install.packages("rbenchmark")  # Install Package
library(rbenchmark)             # Load Benchmark Package
benchmark(chol(a))              # Benchmark 100 times
```

Results (Mid-2009 MBP, 2.53 GHz duo core CPU, 4 GB RAM):

Test	Replications	Elapsed	Relative	User Self	System Self
vecLib	100	136.512	1	249.465	5.402
default	100	564.854	4.138	562.928	1.759

macOS: Revert to single-threaded BLAS

To *revert* to single-threaded BLAS (default) use:

```
# Open terminal and type the following:
Line 1: cd /Library/Frameworks/R.framework/Resources/lib

Line 2: ln -sf libRblas.0.dylib libRblas.dylib
```