ILLINOIS

# Intro to Parallel Computing and Big(ger) Data

## Lecture 15

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

External reads

- Shorter tutorial on *Parallel Computing in R* by Matt Jones
- Comprehensive tutorial by Jonathan Dursi (with sources)
- Parallel Computing for Data Science book by Norm Matloff
- CRAN Task View on High-Performance Computing

## R: Our engine

- R itself is “single-threaded”
- That means a single *thread* of execution in our program
- (There are some exceptions but this broadly holds)
- The running program is generally refered to as a process
- The internal design of R makes this hard / impossible to change
- So this puts a limit on how much work we can do “at once”

Yet ...

- Computers have become multi-core
- Even laptops and cell phone have four or more cores
- Servers can have dozens of cores (and more than one cpu)
- And then we can of course have multiple servers
- Two broad answers:
  - *implicit* parallel work on the same computer (our focus here)
  - *explicit* parallel work across different computers (not here)

## Parallel work on the same computer

- Our running process may *fork* other processes
- Our (single-threaded) process may contain multi-threaded parts
  - generally well-shielded subroutines with well-defined entry
  - R itself has a few multithreaded helper functions
  - `data.table` is an example: many parallelised routines
  - this often involved multithreaded C/C++ code
  - using either the `OpenMP` or `phreads` libraries
  - and that is not our focus here
- So focus will be on R tools mostly for multi-process work
- Some platform-dependency, see `parallel` package vignette

## One simple exception

- R can be set up to use parallel Matrix math libraries
- This depends on how R is built / compiled
- You need to check the details on your installation
- The `sessionInfo()` function displays it
- Look for BLAS (== Basic Linear Algebra Subroutine) and LAPACK (== Linear Algebra PACKage)

On my computer here:

```
R> sessionInfo()
R version 3.6.1 (2019-07-05)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 19.04

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.3.5.so
```

Here OpenBLAS signals that such a parallel BLAS/LAPACK implementation which in fact multithreaded so *e.g.* a matrix multiplication can multiply and some several row/column combinations at one.

On our RStudio Cloud instance:

```
> sessionInfo()
R version 3.5.3 (2019-03-11)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.6 LTS

Matrix products: default
BLAS: /usr/lib/atlas-base/atlas/libblas.so.3.0
LAPACK: /usr/lib/atlas-base/atlas/liblapack.so.3.0
```

This is also a decent choice: ATLAS stands for Automatically Tuned Linear Algrebra Subroutines – optimized, but *not* multithreaded (as the cloud server will likely already be part of a shared larger computer).

## Key Points

- `parallel` is included with R, part of every installation
- Consolidates parts of earlier packages
  - `multicore` which offered easy use of multiple cores
  - `snow` which combines both *implicit* and *explicit* parellism
  - `snow` is still available as an external package
- pdf vignette for package `parallel` is a very good introduction

Two key functions

- `parLapply(cl, x, FUN, ...)` as a parallel `lapply`
  - uses a `cl <- makeCluster(...args...)` as first argument
  - and has to call `stopCluster(cl)`
- `mclapply(X, FUN, ..., mc.cores)` ditto
  - just dispatches to the `mc.cores`
  - no need for `cl` object (easier, less flexible)
  - but `mclapply` is not available on Windows
- uses process parallelism either way

Easy and reliable work-horse

- generalizes `lapply`
- sweeps a function over all elements of a vector or list
- easy to use on single Linux or macOS machine

- Background: every process on Unix has a process id, or ‘pid’
- Tools like `ps` or `top` show it
- R wraps many system functions, here we can use `Sys.getpid()`

```
Sys.getpid()
```

```
## [1] 2601
```

The displayed number will vary, but if you `ps -vax` and `grep` for the same id (on the shell) you should see your R process.

- Now let’s run this six times via `lappply`
- We immediatly collapse the result into a vector for compactness
- The function used by `lapply` must take an argument so we wrap `pid()`

```
do.call(c, lapply(1:6, function(x) Sys.getpid()))
```

```
## [1] 2601 2601 2601 2601 2601 2601
```

Six times the same value. Do you know why?

- Now let’s run this six times via `mclappply`
- We immediatly collapse the result into a vector for compactness

```
suppressMessages(library(parallel))
do.call(c, mclapply(1:6, function(x) Sys.getpid()))
```

```
## [1] 2649 2650 2651 2652 2649 2650
```

Four different values, two repeats. Can you think of a reason why?

# 'Simple' mclapply example

ILLINOIS

- We just illustrated process-parallel operations
- `lapply` ran sequentially in the same R process:
  - same process id each time
  - (though random across different R sessions)
- `mclapply` spawned different processes
  - different process id values

# Bootstrap Example: Cf Section 10.1 of 'Parallel'

ILLINOIS

```
suppressMessages(library(boot))
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
cd4.boot <- boot(cd4, corr, R = 999, sim = "parametric",
                 ran.gen = cd4.rg, mle = cd4.mle)
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 999 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = cd4.boot, conf = 0.9, type = c("norm", "basic",
##     "perc"), h = atanh, hinv = tanh)
##
## Intervals :
## Level      Normal              Basic              Percentile
## 90%   ( 0.4528,  0.8593 )   ( 0.4565,  0.8644 )   ( 0.4761,  0.8706 )
## Calculations on Transformed Scale;  Intervals on Original Scale
```

# Bootstrap Example: Cf Section 10.1 of 'Parallel'

ILLINOIS

```
suppressMessages(library(parallel))
mc <- 4 # adjusted from 2, also adjuested R on next line
run1 <- function(...) boot(cd4, corr, R = 250, sim = "parametric",
                           ran.gen = cd4.rg, mle = cd4.mle)
set.seed(123, "L'Ecuyer")  # To make this reproducible
cd4.boot <- do.call(c, mclapply(seq_len(mc), run1) )
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)


## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = cd4.boot, conf = 0.9, type = c("norm", "basic",
##     "perc"), h = atanh, hinv = tanh)
##
## Intervals :
## Level      Normal              Basic              Percentile
## 90%   ( 0.4569,  0.8553 )   ( 0.4549,  0.8565 )   ( 0.4995,  0.8711 )
## Calculations on Transformed Scale;  Intervals on Original Scale
```

Creating two functions:

```
bstrpSer <- function() {
    cd4.boot <- boot(cd4, corr, R = 999, sim = "parametric",
                     ran.gen = cd4.rg, mle = cd4.mle)
    boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
            conf = 0.9, h = atanh, hinv = tanh)
}
bstrpPar <- function() {
    run1 <- function(...) boot(cd4, corr, R = 250, sim = "parametric",
                               ran.gen = cd4.rg, mle = cd4.mle)
    cd4.boot <- do.call(c, mclapply(seq_len(mc), run1) )
    boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
            conf = 0.9, h = atanh, hinv = tanh)
}
```

And timing them:

```
library(rbenchmark)
benchmark(bstrpSer(), bstrpPar(), replications=100)[,1:4]
```

```
##          test replications elapsed relative
## 2 bstrpPar()          100   3.620    1.000
## 1 bstrpSer()          100   6.084    1.681
```

so here we see a 61% timing difference on the simple example.

Important lesson: speedup from parallel task almost always sublinear as overhead involved.

## `foreach`

- The `foreach` package is popular and widely used
- It uses an 'iterator' approach to 'loop' over a set of loop indices
- Different backends can be registered
- This makes it easy to switch between serial and parallel modes

ILLINOIS

## Standard `for` loop

```
for (i in 1:3) print(sqrt(i))
```

```
## [1] 1
## [1] 1.41421
## [1] 1.73205
```

## Using `foreach` in serial mode

```
library(foreach)
rl <- foreach (i=1:3) %do% sqrt(i)
do.call(c, rl)  # list into vector
```

```
## [1] 1.00000 1.41421 1.73205
```

- `%do%` operates in the object created by `foreach`
- a `{ ... }` block can follow as well as single call seen here

`foreach` and `%doParallel%`

- Several different backends available
- One of the simplest is `doParallel`
- We register a backend, and use `%dopar` instead of `%do`

## foreach and %doParallel%

```
suppressMessages(library(doParallel))
cl <- makeCluster(4)
registerDoParallel(cl)  # use multicore-style forking
rl <- foreach (i=1:3) %dopar% sqrt(i)
do.call(c, rl)
```

```
## [1] 1.00000 1.41421 1.73205
```

```
stopCluster(cl)  # cleanup is good practice
```

- New(er) package (family), may replace `foreach` use over time
- Many interesting features but also easy *parallel* use:

```
# two functions from package 'future'
plan(multiprocess)
# function from package 'future.apply'
rl <- future_lapply(countries, FUN=doWork)
res <- do.call(rbind, rl)  # combine into single list
```

Registers for multiprocess use (on same machine), applies `doWork` across counties and combines results

# Notes

## General comments

- We want jobs 'large enough' to be worthwhile running in parallel
- Overhead in coordinating runs so using 2,3,4,... threads or cores or machines generally does *not* scale 2,3,4,... times
- Many tasks are 'embarrassingly parallel':
  - Bootstrap and Monte Carlos simulation
  - Markov Chain Monte Carlo and Gibbs sampling
  - Machine Learning calibration of hyper-parameters
  - Machine Learning crossvalidation across distinct data sets
  - Data parallelism: different days or regions or sources ...

## General comments

- Key is not to have dependence between data sets
- `mclapply()` easy to use and great workhorse
- `future` package (and `future.apply` etc) another (newer) excellent candidate

R prefers data in its memory

- That is a constraint and “feature” for exploratory analysis
- But because of “big data” alternatives exist (see next slide)
- A lot of this is too specific and requires special setup
- Just know it is there so you can catch up when the need arises

## R Packages that help

- `biglm` can run regressions etc out of memory
- `bigmemory` keeps data in RAM but outside of R
  (this avoids extra copies)
- additional packages build on top of `bigmemory` for analysis
- `disk.frame` uses `fst` to treat data as it were local
- computation inside the database (*i.e.* `dplyr` and `dbplyr`)
- and of course R as part of large batch systems (Spark, Hadoop)
- a bit more on Big Data is at this CRAN Task View

Simple example using `iris`

```
library(fst)
# generate a sample fst filea
path <- paste0(tempfile(), ".fst")
# write iris data to file
write_fst(iris, path)
```

```
ft <- fst(path)    # create a fst_table object that can be used as a data frame
print(ft)          # print snapshot view
```

```
## <fst file>
## 150 rows, 5 columns (file16b86eb661d8.fst)
##
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##         <double>    <double>     <double>    <double>  <factor>
## 1            5.1         3.5          1.4         0.2    setosa
## 2            4.9         3.0          1.4         0.2    setosa
## 3            4.7         3.2          1.3         0.2    setosa
## 4            4.6         3.1          1.5         0.2    setosa
## 5            5.0         3.6          1.4         0.2    setosa
## --            --          --           --          --        --
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

Some more code examples – not executed, see `example(fst)`

```
# select columns and rows
x <- ft[10:14, c("Petal.Width", "Species")]

# use the common list interface as with data.frame
ft[TRUE]
ft[c(TRUE, FALSE)]
ft[["Sepal.Length"]]
ft$Petal.Length

# use data frame generics
nrow(ft)
dim(ft)
colnames(ft)
rownames(ft)
```

# Summary

### Parallel Programming in R

- Multithreading at C/C++ source level (for experts)
- Multiprocess via packages like `parallel` and `future`
- Switching from `lapply` to `mclappy` is very easy
- `future` package best path forward

### Bigger Data

- Several solutions via packages
- New package `fst` looks very promising