Big Data and Parallel Work with R

What We'll Cover

- Data Limits in R
- Optional Data packages
- Optional Function packages
- Going parallel
- Deciding what to do

Data Limits in R

Big Data?

- What is big data?
- More and more often, we have GB or TB of data we need to process
- Reaching physical limits for space and time
- Need to find solutions

Limits

- Space is limited to dozens of GB
- Time is limited by human patience
- Most installations of R use 32-bit versions of libraries, so have limits like 2^{31}-1, or ~2 billion matrix elements
- To use more than this, usually needs end user to compile both R and external libraries (like BLAS)

Solutions

- Use more efficient storage schemes to make maximum use of available space
- Use incremental algorithms to do calculations on data chunks
- Use parallel machines to divide the data across multiple CPU's or machines

Optional Data Packages

Why?

- When dealing with big data sets, need to work with them efficiently
- Problem with regular lists is that every element needs to be checked before being worked with
- The simplest move is to use data.frame
- Frames have some restrictions
  - Data within a column needs to be all the same type
  - Rows need to be the same size

Data Frames

- read.table returns a data frame
- data.frame() allows you to create a data frame from other data constructs
- Columns need to be named
- Rows need to be named, names can be one of the columns
- read.table can be told to read in the data incrementally

NetCDF

- In scientific computing, data often comes in NetCDF format
- The R package to handle NetCDF allows for files to be opened but not loaded
- You can then incrementally access the data without using up all of the available memory

Databases

- Databases provide a way of storing massive amounts of data and to be able to pull out selections of data
- R packages provide access to all standard database engines (MySQL, Oracle, Postgres, etc.)
- In parallel environments, there are packages to use Hadoop

bigmemory

- The package bigmemory provides for multi-GB data sets
- Shared-memory can be used by multiple processes on the same box
- File-backed access can be used, which aids multi-machine access
- The core object is big.matrix

big.matrix

- Implemented in C++
- The standard big.matrix uses RAM, and so is limited
- filebacked.big.matrix uses the hard drive
- A big.matrix is handed by reference to functions, not by value, so there may be side-effects

Optional Function Packages

Functions

- Dealing with large sets of data requires efficient functions
- The simple solution is to use functions like apply
- The bigmemory project also offers biglm and biganalytics

lapply

- With large data sets, you may need to apply some function to each value in a list
- To do this, you can use the function lapply

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,
FALSE,TRUE))
# compute the list mean for each list element
lapply(x,mean)
```

biganalytics

- The bigmemory project allows provides functions optimized for using big.matrix
- It adds overloaded versions of the standard descriptive statistics functions (mean, var, etc)
- It also provides an overloaded version of apply

biglm

- If you are trying to fit a model to a large data set, you can use the biglm package
- Introduces a biglm function
- You can step through additional chunks of data with the update function

Going Parallel

Parallelization?

- For large problems, it may make sense to use multiple CPUs
- Traditional methods use packages like multicore, SNOW, Rmpi, etc
- Starting with version 2.14.0, R includes the package parallel
  - implements multicore
  - implements SNOW
  - provides parallel RNGs

Parallel

- To use the parallel package
  - library("parallel")
- The multicore part is implemented on most systems using threads
- On Windows, this gets implemented as separate processes
- Used by functions like mclapply
- The default is to break the list into even chunks
- You can break into smaller chunks to aid load balancing

SNOW

- Simple Network Of Workstations
- Involves creating a cluster of processes to use
- These processes can be on one machine or many networked together
- There are multiple steps involved

Clusters

```
library("parallel")
cl <- makeCluster(size)
parLapply(cl, list, FUN)
stopCluster(cl)
```

Clusters - 2

- If you want to use shared memory processes, you can use makeForkCluster
  - This doesn't work on Windows
- If you want to use multiple machines, you can use makePSOCKcluster
  - This uses ssh on most machines
  - On Windows machines, you want to use something like rshcmd="plink.exe"

foreach

- The foreach function makes loops easy to deal with
- You can force whether it happens serially or in parallel

```
foreach (i=1:3)  %do%
    sqrt(i)

foreach (i=1:3)  %dopar%
    sqrt(i)
```

Deciding what to do

Profiling

- In order to decide what to optimize, you need measurements
- This is called profiling
- You can profile time, space, memory usage, function calls, etc.

Time

- Usually the first thing to do is measure how long things take
- system.time is easiest thing to do

  system.time(expr, gcFirst=TRUE)

- This calls the garbage collector first

Memory

- Need to see how much space things are taking
- In general, you can use
  - memory.profile()
- For the size of a specific object, use
  - object.size(obj)
- You can force a garbage collection with
  - gc()

General Profiling

- There is a full set profiling functions

```
Rprof(filename="profile.log", append=FALSE, interval=0.02, memory.profiling=FALSE)
...
...
Rprof()
summaryRprof("profile.log")
```