POIR 613: Measurement Models and Statistical Computing

Pablo Barberá

School of International Relations
University of Southern California
`pablobarbera.com`

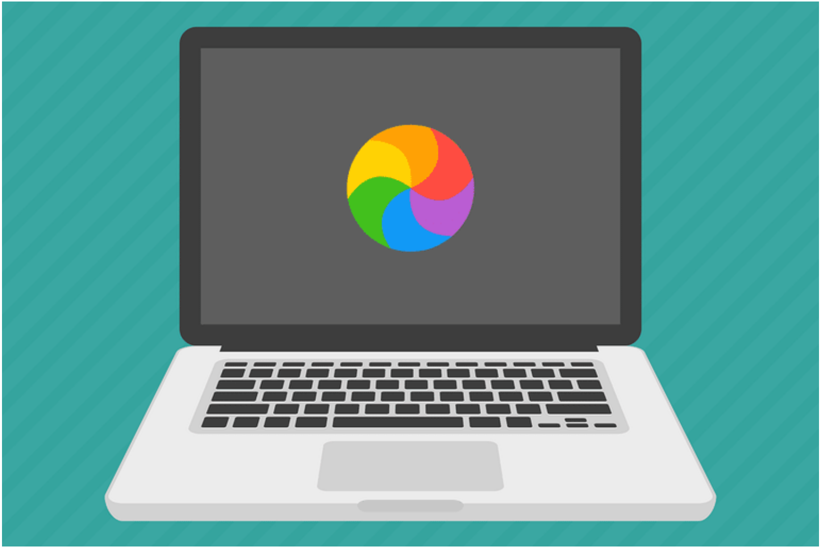
Course website:

pablobarbera.com/POIR613/

Today

1. Solutions for last week's challenge
2. Discussion: experimental research in the digital age
3. Efficient programming in R
4. Parallel computing

Efficient data analysis with R



Myths about R as programming language

1. R is an interpreted language, so it must be slow
   - Interpreted = executes code directly without compiling
   - Compiled code = code executed natively on CPU (fast!)
   - BUT: many functions are written in C and C++ and thus run in fast machine code
   - Slow code can be written more efficiently
2. All objects in R are stored in memory
   - You cannot open datasets larger than RAM
   - BUT: most laptops now have 8+ GB of RAM (+virtual mem)
   - `bigmemory` package: work with files on disk
   - Easy to work with large databases in the cloud
3. R only uses one core of your CPU
   - Unlike STATA, no multi-core computing out of the box
   - BUT: many functions and packages now take advantage of multi-core computers
   - Easy to write your own code to do parallel computing

My data is too big! My code is too slow!

What to do?

1. Buy a better computer or expand RAM memory
2. Write more efficient code
3. Use parallel computing
4. Move your code/data to the cloud
5. Use out-of-memory storage: SQL databases, bigmemory package, Hadoop...

Writing efficient R code (Part I)

- Conventional wisdom: avoid for loops at all costs!
- But simply rewriting loops will not make code faster
- Key: use vectorized functions instead of loops

  See `day1/01-efficient-programming.html`
- What is slowing our code down?
  - Additional function calls: `for`, `:`, `[`, `<-`
  - `sapply` hides explicit loop, but loop is still there, and implemented in R code
- Why was + so fast? Implements vectorization by vector filtering
  - Takes vector as input and return vector as output
  - Loop is done in machine native code
  - Other vectorized functions: `ifelse()`, `which()`, `rowSums()`, `colSums()`, `sum()`, `any()`, `rnorm()`...

Writing efficient R code (Part II)

- A common bottleneck is memory re-allocation, e.g.:

```
result <- c()
for (i in 1:n){
   result[i] <- x[i] + y[i]
}
```

- In iteration, `R` re-sizes the vector and re-allocates memory
- For large operations (e.g. data frames), this can make your code really slow
- Solution: pre-allocate vector size:

```
result <- rep(NA, n)
for (i in 1:n){
   result[i] <- x[i] + y[i]
}
```

- See `day1/01-efficient-programming.html` for more examples

Parallel computing

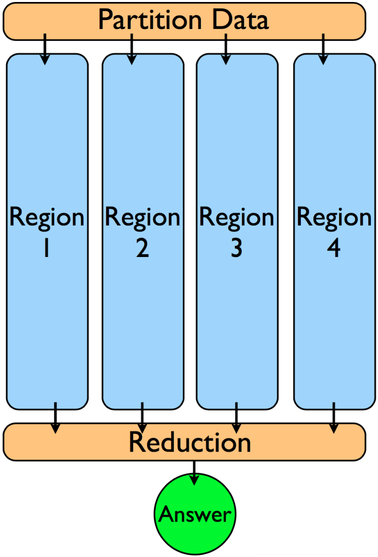
Some hardware terms:

- Node: a single motherboard, with possibly multiple processors
- Processor: silicon containing one or more cores
- Core: unit of computation
- Most modern CPUs (processors) have multiple cores

Logic of parallel computing

Split-apply-combine framework (Hadley Wickham and others):

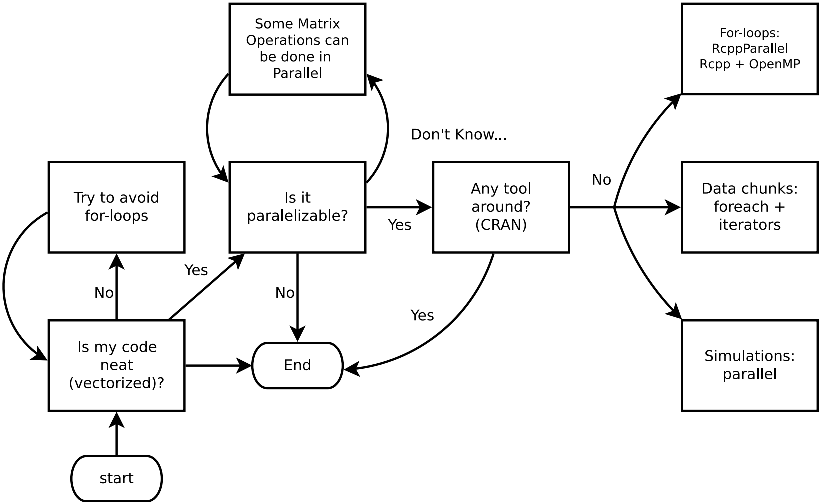
- Split your code and data across multiple nodes/processors/cores
- Apply computation in each region
- Combine the individual results into an aggregate answer



Logic of parallel computing

- BUT: overhead (e.g. splitting and combining data also take some time, no free lunch!)
- Works best with embarrassingly parallel problems:
  - Statistical simulation using multiple seeds
  - Word counts in documents
  - Cross-validation or ensemble learning
  - Rule-of-thumb: can you change the order of the iterations without altering the result?
- Sometimes problematic: applying on subsets of data, or when full dataset is needed in each node
- Not parallelizable: Markov-Chain Monte-Carlo methods, cumulative sums, etc.

Parallel computing



Source: Vega Yon and Garrett Weaver, 2017

Parallel computing in R

Two main approaches:

1. R packages
   - `parallel`: built-in package with support for parallel computation, including random-number generation (good for statistical simulation)
   - `foreach`: new type of loops that supports parallel execution (good for data analysis)
   - `iterators`: tools for iterating over various R data structures (more advanced)
2. Running C++ code in R:
   - `RcppArmadillo`: interact with C++ linear algebra library
   - `OpenMP`: utility to improve multiprocessing using shared memory; works across all platforms

And many others (e.g. Spark, Hadoop, RcppParallel...) we will not cover in this course. See the High-Performance and Parallel Computing Task View

For more: see slides+code by Vega Yon and Garrett Weaver