Optimizing your simulations lecture

Basic idea behind optimization

- Make your code run faster
- Not just for convenience
- Luria del sim takes 1.62 secs per loop
    - 1000x is 1620 secs or 27 minutes
    - See mu r and n - 20 values of each
    - 20 × 20 × 20 × 27 minutes = 216,000, or 150 days
    - Takes five months

Computer science detour: big o notation

- Use to classify algorithms by how their space or time reqs grow with input size
- Number of iterations for our sim is o(n)
    - Grows linearly
    - That would change if we ran sims in parallel
- Also think of how different subprocesses scale with input
    - Growth of cells in o(2^n)
    - Could this be o(n) or even o(1)?

how do we make it go faster?

There are four main ways to make code faster, which vary in how easy or hard they are

- Using a faster language has the biggest payoff, but it can be difficult to learn if you dont already know - a good option for people that are serious about comp. Bio
- Using a faster library is an alternative, but can have minimal pay off
- Refactoring your code is the meat and potatoes of optimization
    - But Can be a deep rabbit hole of algorithms and computer science
- Parallelization is a fast option to get things to run fast, but requires added complexity

First let's talk about how the code is actually run
I'm not at expert
And to do that we have to first talk about how computers work - not necessary but fun

- Each but is stored in a transistor
    - Old fashion tube or relay
- Each one stores a bit - a 1 or 0
- Transistors can be combined to make logic gates eg and or Xor
- Logic gates combine to be adders, subtractors, multipliers, etc
- Ergo calculator

At base level, when the computer runs your code, it takes a piece of information (binary number, string, etc) from one register (spot where the info was) does some action to it, and puts it in a different register

The set of instructions for the computer to do these actions is called machine code

- At the basic level, every piece of code is run in machine code

When you write a piece of code, it gets converted to machine code
- This is done by a compiler or interpreter
- Both Python and r are interpreted languages
    - Makes them interactive
- Analogy: reading a book in foreign language
    - Compiler translates whole book then you read
    - Interpreter translate each line as you get to it
- For some code in R and Python, they are interpreted directly into machine code
- For a lot of other code, it's actually written in c++, c, or Fortran

These are called low level languages
- Because they are much closer to machine code than R or Python
- Run wayyyyy faster
- Trade off in readability
    - Coding time vs running time

How to make your code go faster:
- Use a different language
    - C++ is champ in science. Julia is rising star
    - Unrealistic for many people
    - Trade off between writing time and running time
- Other option: use libraries that are written in lower level language
    - Numpy and numba

The next strategy is the most important one, but it's also the hardest: refactor your code
- Refactoring is changing your code without changing external behavior
- Idea is to come up with a way to do the same thing but wayy faster
    - Ie a new algorithm
    - Kinda at the core of a lot of computer science

Some general strategies I can offer
- Understand data structures and how to access them
- Look at how many operations your code does
- Be efficient with memory
- Vectorize everything
- Use pre-optimized functions

Understanding data structures:
- Both Python and R are object oriented languages
    - Means objects have methods
    - Think splitting a string

- Using the correct data structure and methods to store and work on your data save time and memory
    - Data structures are optimized already for diff tasks
- Example: accessing values by name: us a dictionary
    - Accessing values by position: use a list
    - Only need unique values: use a set
- Big one in Python, setting a value in an array is wayyyyyy faster than appending to a list

Look how many operations your code is doing

- Each step takes time
- The number of operations relates to big o
- Example: chain of if statements? Break out of it after you the condition was met
- Example: for for loops, are you calculating the same value over and over again

Be efficient with memory

- Don't needlessly store variables
- Understand what makes a copy vs a view
- For large files, only reading part to conserve memory
    - Dask library
- Storing data in binary format
    - Pickle library in Python

Vectorize everything!

- Vectorizing is the process of taking a for loop and using a function to do the same thing
- Often biggest improvement is from using linear algebra
    - Doing math is most basic for computers, so it's fast
- Example: multiplying elements from two arrays, number of operations is far lower with vectorization
- In general avoid for-loops like the plague
    - Loop infrastructure is expensive
- Many vectorization functions for math already built in and optimized
    - Other non-math functions vectorized too
- For more complicated functions, use map or apply in Python, or apply family in Python
    - Beware tho, if your function is very complicated, vectorizing functions end up just doing a forloop anyways, negating any benefit

Lastly, use pre built functions

- People have spent the time optimizing code so you don't have to

So that's refactoring, the last thing to talk about is paralleling code

Parallelizing code works great if you're doing the same thing over and over again AND each time is independent

For example, the running our 8 million simulations from the example at the beginning is a prime candidate for parallelizing code

There are two levels at which this can be accomplished

- First is parallel processing done locally
- The second is using a computing cluster to run several jobs at the same time

Parallel processing is basically running lines of code in parallel on the different processors or cores in your computer

- Most modern computers will come with multiple cores in the CPU
- Foreach in R, multiprocess in python
- Generally, if you're doing something heavy this isn't ideal because it will keep your computer busy
    - And you only have so much processing power
- Big caveat: the infrastructure for parallel processing takes time!! Sometimes it can make your code much slower instead of faster

Take advantage of our very expensive and very powerful computing cluster!

- Really the best way to get a lot of code run once you've optimized it as much as you reasonably can
- Jobs are scheduled with slurm
- Submitted with sbatch scripts

some games you can play to get your jobs running

- Look at see which nodes are busy, use others
    - Squeue
- Request minimal resources - priority given based of requests
    - Only as much time, cores, and memory as your job needs
- Launch a job to submit your jobs
    - Minimal memory job
    - Use squeue within a while loop to submit a few jobs at a time
- For more complicated jobs
    - Write a Python/R script to write sbatch commands and launch them
- Combine repeated jobs to save on set up time
    - Launching a job, writing error and output files, loading required packages etc all takes time
    - Normally it's fastest to have a single script doing multiple iterations, then running that script multiple times in parallel on the cluster