Make Python Go "Brrrrrr"

Joseph Long
March 22, 2022

Concepts

"embarrassingly parallel" problems

- Can be subdivided into units that do not require communication between workers
- Example: take a room full of people and ask them to add up the digits in their birth year and write it on a slip of paper
  - They have the input information, which *may* be different for each worker
  - They do not need to ask their neighbor for any information
  - They can all write their result at once (as long as they all brought their own pen)

Concepts
"embarrassingly parallel" problems

- Astronomy examples:
  - testing orbit fits with different parameters
  - calibrating the contrast at many points in an image with fake planet injections
  - dark/bias/flat correcting a whole night of observations
- "different parameters", "many points", "observations"... all subproblems that can be computed independently

Concepts

non-"embarrassingly parallel" problems

- Cannot be subdivided into units that do not require communication between workers
- Example: take a classroom and have every row calculate the sum of everyone's birth year in the row and write it on the chalkboard
  - They need input information from their neighbor to proceed
    - Go linearly along the row, passing to the right? Some other arrangement?
  - If there's only one piece of chalk, they will also have to wait patiently to write their result when they finish

Concepts

non-"embarrassingly parallel" problems

- Astronomy examples
  - N-body simulations
    - every particle can influence every other particle
  - fluid dynamics simulations
    - Fluid in one simulation volume interacts across the boundary with other volumes
  - file compression (with some caveats)
    - Need to know the repeated sequences from the whole file to compress efficiently

Concepts

pure functions vs. functions with side effects

- Program functions are rather different from math functions like f(x) = 2x
  - Every time you evaluate f(10) you will get 20, no exceptions
  - The only thing that happens when you evaluate f(10) is a multiplication by 2
- Program functions can be "impure" when they cause side-effects
  - ```
    def f(x):
        print("called f(x)")
        return 2 * x
    ```
  - Spot the side effect
- Know your arguments! What about global state?
    ```

The first kind of problem

- Your Python code runs slowly
- You know it is possible to parallelize
  - For example: You evaluate 10,000 points by running the same set of steps on each one, and the value at one point doesn't depend on the values of the others
- You have more than one processor core, but Python's only using one while the others sit idle

The first kind of problem

- You have a bunch of cores in your laptop sitting idle while your Python program maxes out 1 core
- OR your Python code uses multiprocessing or parallel linear algebra libraries to use all the cores on your computer, but...
- It's still not fast enough, and you'd like to run it on multiple **computers** in parallel

~~The~~ A Solution

- Ray is a Python package for parallelism written on a base of interesting technologies
- The interface you deal with allows you to ignore almost all the details of what goes on under the hood
- The same interface will let your code farm out work to multiple cores, or to multiple computers, without any code changes

Demonstration

Caveats

- Cluster-scale Ray is hard to set up the first time (but remarkably solid since)
- Ray functions **only** run in the ray executor, making automated tests awkward
  - I usually have def _foo(arg1, arg2) and then foo = ray.remote(_foo) so I retain the ability to call _foo directly if I need to.
  - Structuring your code so that @ray.remote functions are the "glue" and your analysis is actually implemented as collection of regular Python functions is wise
- Visibility (logging, tracing, debugging) grows unavoidable complexity when running parallel or distributed code.
  - Test at small scales first.

The second kind of problem

- You already know to avoid loops in your code when you can use NumPy array operations instead
- Some things are still just too slow (or, equivalently, called so many times it adds up) and you need to speed them up
- Example: image interpolation by bicubic convolution
  - retrieve 4 pixel values 4 times and evaluate the final interpolated location
  - NumPy doesn't really speed up N=4 loops

~~The~~ A Solution

- **Numba** is an optimizing JIT compiler for Python that understands NumPy operations natively
  - Appreciate how incredibly tricky this is
  - Computer scientists bravely implement compilers so the rest of us can live in peace
- Apply one decorator, get instant performance*
  - *Terms and conditions apply.

Demonstration

Caveats

- Overusing @njit and @jit can, in some cases, **slow down** your program
  - The first time your code is evaluated, it will invoke an optimizing compiler... which is a pretty complex piece of code designed to eke out the maximum performance from a program
- Calling back and forth between @njit and regular Python functions can be tricky
- Numba itself isn't available everywhere Python is, resulting in occasional compatibility problems