Lab #14

Physics 91SI Spring 2013

Objective: In this lab, you will use Cython and C to speed up an implementation of John Conway's *The Game of Life* in Python.

As usual, log on to `corn` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab14 lab14
```

The repository contains several source code files, as well as a file called `setup.py`. As demonstrated in lecture, this file can be used to generate automatically an executable python module from the present Cython and C files. You can call `setup.py` either from the shell or from inside IPython:

From the shell: `./setup.py build_ext --inplace`
Inside IPython: `run setup.py build_ext --inplace`

This will take most current version of your C and Cython files and compile them into a Python module, `life_cy`, that you can then import just like any other Python library. (Note: It will create the files `life_cy.c` and `life_cy.so` and the directory `build`.)

Remember that you have to use this command every time you modify the C (`.c`) or Cython (`.pyx`) files in the folder; otherwise, you won't have the newest version compiled. If you're inside of IPython, make sure to call `reload(life)` and `reload(life.life_cy)` if necessary.

Also, remember to `hg commit` often to save your changes, and submit your code the end of class.

Part 0: The Game of Life

In this part of the lab, you should briefly familiarize yourself with the algorithms and the Big-Oh scaling of *The Game of Life.* Don't spend more than 10 minutes on this part.

The Game of Life, or simply *Life,* provides a toy simulation of population dynamics. The program consists of a "board" (a 2D grid) of cells. In our version, each cell is either alive or dead, signified by the respective point in the grid containing either a 0 or a 1. In the code, the board is a 2D array of `long`s. The game evolves in discrete time steps. Every step, the following rules are applied:

- Any cell that is *dead* and has exactly 3 neighbors is set to *alive* (reproduction)
- Any cell that is *alive* and has 2 or 3 neighbors survives (stays alive)
- Any cell that is *alive* and has more than 3 or less than 2 neighbors *dies* (overpopulation and underpopulation, respectively)

Take a look at the file `life.py`. Familiarize yourself with the algorithm and convince yourself that it is doing the right thing. What is the scaling of the algorithm in N?

The `life()` function starts the actual simulation. Read the comments to figure out what arguments it takes. Be aware that the `"c"` and `"cython"` versions of the algorithm are not implemented yet. It will be your job to implement them later.

You can try out the simulation as follows: open up an IPython session and type at the command line:

```
import life
life.life(40, 0.2, plotting=True, version="python")
```

You should be able to watch the simulation. Now use the `%timeit` magic command to time the execution:

```
%timeit life.life(40, 0.2, plotting=False, version="python")
```

Note that we have **set `plotting` to `False`.** This is because the plotting over the ssh X-server takes a long time and outweighs the actual execution time. Try it out for different N and make sure the scaling is what you expected it to be.

Part 1: Life in Cython

In this part, you'll rewrite the `update()` function in Cython and observe the speedup that comes with this simple change in the code. In order to keep the old Python version, you will write this new function in the file `life_cy.pyx`. All function calls and definitions have already been written; all you have to do is fill in the body for `update_cy()` and `getNeighbors()`. Make use of all of the tricks that you learned about in lecture to maximize the performance of those two functions. They won't look very different from the old Python versions, but you should be able to get at least a factor of 25 speedup with some simple modifications. Once you have written the two functions, first run the simulation in `"cython"` mode without `%timeit` with `plotting` set to `True`. Does it behave correctly? If so, use the `%timeit` command with `plotting` set to `False` and compare the performance when `version` is set to `"python"` to when it is set to `"cython"`. What speedup do you get?

Part 2: Life in C

Now it is time to maximize the performance of the *Life* simulation. Notice that the Cython file `life_cy.pyx` is set up to call a pure C function, called `update_pure_c()`, through the wrapper function `update_c()`. It is now your job to go to the file `life.c` and implement the `update_pure_c()` and `getNeighbor()` functions in pure C. Once you have written the two functions, first run the simulation in `"c"` mode without `%timeit` with `plotting` set to `True`. Does it behave correctly? If so, go ahead and time the execution without plotting for all three versions. By writing straightforward C code, you should be able to get a relative speedup of over 400 over the pure Python version.