Python Optimization and Multiprocessing Lab

Part I. Optimization

Project 1

Copy the file slowcode.py from /share/resources/tutorials/optim on Rivanna and profile it. What might you do to increase the speed? The slowcode.py file will need the input file slow_py_data.csv

Project 2

Download the code original.py from the same location as Project 1. Make as much improvement as you can. Be sure to keep a log of timing data, along with all changes you make. Some changes may actually slow down the code, especially for Python.

Python: this can be a challenging exercise. If you have not improved the code significantly in an hour or so, move on. Hints: study the documentation for NumPy to find as many "vectorized" built-ins as you can. Try out Numba if you have time.

Input files input-<number>.zip are supplied. Unzip on the cluster with
gunzip input-<number>.zip
The number is the number of coordinate sets for "molecules."

Please examine the code and note that it requires two parameters input from the command line. One parameter is the "radius" parameter. We suggest using 0.9 for that.

For Python, we strongly advise that you use the smallest number of "molecules" for input. For compiled languages you will probably need to use the largest number to run long enough to see any changes.

Part II. Multiprocessing

Project 3

One way to compute pi is to use numerical integration. Several integral formulations exist but a simple one involves $\int dx/(1+x^2)$. In pseudocode the solution can be expressed as

```
Nsteps=Something
h=1./(double)Nsteps
for i=1 to Nsteps do
   x=(i-0.5)*h
   total=total+4./(1.+x*x)
end
pi=h*total
```

Write a program to compute pi. Run it in serial for a relatively small Nsteps to make sure it works. When you are done, parallelize it with OpenMP. Add the timing functions to return total time required. When you are satisfied that it is working, increase the number of steps to 100000000. Run with 1, 2, 4, and 8 processes and plot the scaling curve.

Project 4

You should use the cluster for this exercise. You may write and test your program on the frontend, but submit runs to the compute nodes. Use the following three SLURM options in your script:

#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c <m>
along with other appropriate directives. The value for m will be specified for each run.

We are going to find the maximum of a 3-d surface by "brute force" evaluation of x, y, z values. This is a method of optimization that has become increasingly popular since it is easily parallelizable. The surface is defined by the following rules:

$$\mu_1 = \sqrt{2}$$

$$\mu_2 = \sqrt{\pi}$$

$$\sigma_1 = 3.1$$

$$\sigma_2 = 1.4$$

$$z_1 = 0.1 \sin(x) \sin(xy)$$

$$a = \frac{(x - \mu_1)^2}{2\sigma_1^2}$$

$$b = \frac{(y - \mu_2)^2}{2\sigma_2^2}$$

$$z_2 = \frac{e^{-(a+b)}}{\sigma_1 \sigma_2 \sqrt{2\pi}}$$

$$z = z_1 + z_2$$

The surface is defined over the ranges

$-10\pi \leq x \leq 10\pi$

And

$-10\pi \leq y \leq 10\pi$

Generate a list of N random values for each of x and y over the above range. For testing you can use N=800000. Be sure to measure the time.

Hints: use Numpy. Look up numpy.random.uniform(arglist) to find out how to generate the values. Another hint: for best performance use the numpy built-in `split` to divide an array among the processes.

Print the final value of the maximum you found. If you have time, you can go back and figure out how to return the corresponding x and y values as well.

Once your code is working in parallel, write a SLURM script to run your program on the cluster compute nodes. Use N=10000000 and run one job with 1 process, another with 2 processes, then 4 processes and finally 8 processes. You can launch them all at the same time if you wish. Make a plot of the running time versus the number of processes used. On the same plot show a line for perfect scaling (1/2 the serial time, 1/4 the time, 1/8). How well does your code scale?

For SLURM be sure to specify the number of cores through os.getenv, i.e.

```
ncpus=os.getenv("SLURM_NTASKS")
```

Part III. PyCUDA

Examine the tutorial at https://documen.tician.de/pycuda/tutorial.html
Work through as much as you can.

To run this on a GPU, you will need to log in to FastX Web, or through a shell to rivanna-gpu.hpc.virginia.edu

To submit to SLURM use

#SBATCH -p gpu
#SBATCH --gres=gpu

Both are required. You may also go to OpenOnDemand through

rivanna-portal.hpc.virginia.edu

and start a Jupyterlab instance through the Interactive Apps menus. Be sure to select the gpu partition from the dropdown.

Another option for using Python on a GPU is Numba. A numba tutorial is at

https://github.com/ContinuumIO/gtc2017-numba

To use it, if on Rivanna you will need to create an environment

```
conda create numba_env
source activate numba_env
conda install numba cudatoolkit pyculib
```