Assignment 01 - Multiprocessing

DSE 512

Running and submitting your assignment

For this assignment, you will submit on Canvas the path of a directory on ISAAC. That directory will contain at least two files:

- `assignment1.py`: a Python script that prints the output from the following assignments.
- `run.sh`: a Bash script which will activate a conda environment you have set up, then launches assignment1.py

The instructors will run your code to grade the assignment. Your grade will be based on whether you've properly set up your conda environment (20%), whether your run.sh script works as expected (20%), and the output of each exercise is correct (20%). You'll be given partial credit if we can figure out what went wrong with any of these parts.

Please place your code in a subdirectory within the shared area on ISAAC: For example, `/lustre/haven/proj/UTK0150/$USER/assignment1/`

Problem 1

Using the multiprocessing module, write a simple python program as follows:

1. Create a pool of workers to run parallel tasks.
2. The pool size should be 4.
3. Write a simple function to be run in parallel, call it `def my_pid(x):`. The function should receive as input an integer `x` identifying the task. When called, the function will print to the screen a message in the form: "Hi, I'm worker *ID* (with *PID*)" Where *ID* should be replaced with the argument `x` and *PID* with the process ID of the running worker. **Note: the PID of the current Python process can be found using the `getpid()` function in the `os` module.**
4. Run tasks in parallel using the map function, for arguments `x` ranging from 0 through 9.

Some remarks (**these will not be graded**):

1. Notice that task invocation is not necessarily in-order. PIDs and their order will change if you re-run the program.
2. You can change the call to the `map` function and split the jobs into different size "chunks". See the documentation here: https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool.map. Hint: try adding another argument, `3`, to your `map` call and observe a difference in the PID pattern.

Problem 2

It is always easy to start with something tractable. Here we will compute an approximation to π from one of the many available series:

$$\pi(N) = \frac{4}{N} \sum_{i=1}^{N} \frac{1}{1 + \left(\frac{i-0.5}{N}\right)^2}.$$

The accuracy increases with larger N, which denotes the number of terms to include in the expansion. For the rest of the exercise, you may refer to "exact" π value as returned from NumPy:

```
import numpy as np
print(np.pi)
```

Your goal is to compute an approximation to π using the above formula for different values of N. Using the multiprocessing module, write a Python program as follows:

1. Create a pool of workers to run parallel tasks.
2. The pool size should be 4.
3. Write a function to run in parallel, call it `py_pi`.The function should receive on input a number `N` specifying how many terms to include in the expansion. When called, the function will compute the approximation to π, print the result to the screen along with difference from the "exact" value of π. A typical output can be in the form: "Pi(N) = XXX (Real is XXX, difference is XXX)" Where `N` is the number of terms given as input and each XXX refers to the approximated value, "exact", and difference, respectively.
4. Run tasks in parallel using the `map` function, starting with $N = 10$ for the first worker and increasing 5 times for each subsequent worker until reaching $N = 3906250$ (e.g. N=10,50,250,1250,...).
5. Print the total runtime it took to compute each $\pi(N)$.

Problem 3

The above method of computing an approximation to π is clearly inefficient since work is not evenly distributed between workers. Here we will modify the above procedure such that the expansion will be computed in parallel. Fix N and each worker will get a subset of the expansion to compute. Then, the main caller will collect the results and summarize them to get the final answer.

Using the multiprocessing module, extend the python program from Problem 2 as follows:

1. Write a function to be running in parallel, call it `py_pi_better`. The function should receive on input three parameters: `N`, `i_start`, and `i_stop` that denote the subset the worker should compute from the expansion. For example:

$$\pi(N, i_{start}, i_{stop})_{partial} = \sum_{i=i_{start}}^{i_{stop}} \frac{1}{1 + \left(\frac{i-0.5}{N}\right)^2}.$$

   The function will return the partial result to the caller.
2. Run tasks in parallel using the `map` function, such that the work distribution scheme should be to divide the expansion N as evenly as possible between the workers.
3. After receiving all the results, the caller should add them together and multiply by $4/N$ to get the final answer and print it to the screen as before.

$$\pi(N) = \frac{4}{N} \sum \pi(N, i_{start}, i_{stop})_{partial}$$

   A typical output could be in the form:"Pi(N) = XXX (Real is XXX, diff XXX)" 4.Compute an approximation to π using this procedure for N=100,500,1000,2000,10000,50000 and see if there is any performance benefit compared to trivial version from previous task.

Some additional programming challenges to try on your own (**these will not be graded**):

1. (beginner) Test your CPU performance and code scalability by creating a pool of different sizes (2, 3, 4, more than the actual CPU core count etc.) and see if the overall computation time improves or degrades.
2. (intermediate) Collect results from workers using a queue so the caller stops when the number of items received equals the number of workers.
3. (advanced) Collect results from workers using a simple pythonic list, where for data protection purposes access to the list should be given only after acquiring a lock (don't forget to release it when finished).
4. (advanced) Collect results from workers using a special shared memory array provided by multiprocessing module. You may use whatever scheme to notify the caller that work has ended.