Exercise 1 – multiprocessing Module

Technical Preliminaries

This exercise requires the use of a standard Python 2.7(.X) or Python 3(.X) installation.

You may use your own computer or connect through SSH to the PRACE training cluster.

Instructions to use the PRACE training cluster:

1. For each step below, replace guestXXX with the username you received upon registration.

2. Access the frontend server using SSH with your username and IP: 128.139.196.20

   For Linux/Mac users:
   ```
   # ssh guestXXX@128.139.196.20
   ```
   Enter password when prompted to do so.

   Windows users shall use putty and specify in the connection address:
   guestXXX@128.139.196.20
   Enter password when prompted to do so.

3. At this point you should have a terminal on the training cluster.

4. List available cluster nodes by running:
   ```
   # rocks list host
   ```

   Pick one of the available hosts for your programming task or as designated by the instructor.
   For example, a node name: *compute-0-5*

5. Connect to your node by running:
   ```
   # ssh node_name
   ```

6. The terminal is now attached to the specific computational node. You may run python or any other Linux commands.

7. Python 2.7.9 and 3.6.0 along with latest NumPy are installed at /share/apps/python, but no further configuration should be set for running them.

   To run python 2.7.9 enter at the command prompt:

```
# python
```

To run python 3.6.0 enter at the command prompt:

```
# python3
```

1. A Beginner Task

A skeleton for the program is available online (exe1_beginner.py).

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 as the number of CPU cores available on your node minus 1 (8 cores => pool of 7 workers).

3. Write a function to be running in parallel, call it *my_id*. The function should receive as input the task id.

   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 task number assigned to the worker and PID with the process ID of the running worker.

4. Run tasks in parallel using the *map* function, for a total of tasks equal to twice the number of CPU cores in your node.

Some remarks:

1. Notice that task invocation is not necessarily in-order.
2. You may change the call to *map* function such that it will distribute even work to workers.

2. Intermediate Task

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

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

As you can see, 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)
```

You 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 as the number of CPU cores available on your node minus 1 (8 cores => pool of 7 workers).

3. Write a function to be running 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 error from the “exact” value.

   A typical output can be in the form:
   “Pi(N) = XXX (Real is XXX, diff XXX)”

   Where N is the number of terms given as input and each XXX refers to the approximated value, “exact” and error, 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 $N = 10^{10}$ (e.g. $N = 10, 50, 250, 1250, \dots$).

<u>Some additional remarks:</u>

1. You may also print the total runtime it took to compute $\pi(N)$.

3. Advanced Task

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, modify the above python program 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 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 can be in the form:
   "Pi(N) = XXX (Real is XXX, diff XXX)"

4. Compute an approximation to π using this procedure for $N = 10^5, 10^6, 10^7, 10^8, 10^9, 10^{10}$ and see if there is any performance benefit compared to trivial version from previous task.

Some additional programming challenges:

1. 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. Those with brevity, collect results from workers using a queue so the caller stops when the number of items received equals the number of workers.

3. Those with brevity^2, 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. Those with brevity^4, 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 (there are simple ways to do so).