Exercise 2 – ipyparallel Package

Technical Preliminaries

This exercise requires the use of a recent IPython 5.0 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 ipython or any other Linux commands.

7. IPython supporting 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 IPython compatible with python 2.7.9 enter at the command prompt:


```
# ipython
```

To run IPython compatible with python 3.6.0 enter at the command prompt:

```
# ipython3
```


1. A Beginner Task

This preliminary task is very similar to the previous exercise, just to get started.
All operations should be performed in the interactive prompt.

1. Open a new terminal to your working node.

2. Start a simple controller-engines environment, such that the number of engines is equal to the number of CPU cores available on your node minus 1 (8 cores => 7 engines).

3. Open a new terminal, start an IPython interactive prompt.

4. Import the necessary modules and create a client to connect the controller.
   Verify that the number of reported engines matches your settings.

5. 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 return a string with 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.

6. Run tasks in parallel using the *map* function, for a total of tasks equal to twice the number of CPU cores in your node.
   Watch the returned values to see if results match your environment and invocation.

Some remarks:

1. Notice again that task invocation is not necessarily in-order.

2. Intermediate Task

The goal of this task is to find the minimum/maximum value from a random list.
All operations should be performed in the interactive prompt.

1. Open a new terminal to your working node.

2. Start a simple controller-engines environment, such that the number of engines is equal to the number of CPU cores available on your node minus 1 (8 cores => 7 engines).

3. Open a new terminal, start an IPython interactive prompt.

4. Import the necessary modules and create a client to connect the controller. Verify that the number of reported engines matches your settings.

5. Using NumPy, generate a random array of 10^9 floating point elements from a normal distribution with mean (μ) of -2 and variance (σ^2) of 100.

6. Write a function to be running in parallel, call it *my_minmax*. The function should receive as input the following parameters as a packed argument:
   - array – The full array containing 10^9 random elements
   - i_start – Initial index to start scanning the array from
   - i_stop – Final index to stop scanning the array at (not inclusive)

   When called, the function will find the minimum and maximum values within the requested subset of the array, return that values and also the index of the elements (global index) corresponding to the min/max found.

   For finding the min/max you may use ordinary loops or built-in NumPy functions.

7. Run the computation over the entire array in parallel using the *map* function, so that each engine (task) receives an equal amount of work.

8. Upon return to the caller (interactive prompt), compute and print the final min/max values from the results and also their index.

   An example for such a print:
   "Task (ID) computed in range *i_start*:*i_stop* min = XXX and max = XXX"
   .
   .
   "Final results are min(index) = XXX and max(index) = XXX"

Where i_start is the initial index and i_stop is the last (not-inclusive) index given to the specific reporting task. Replace XXX with the relevant result.

Some additional remarks and programming challenges:

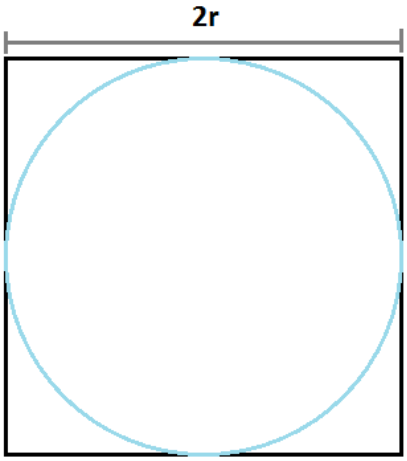
1. You may also print the total runtime it took to compute the min/max in parallel.
2. Compare it with a serial version that you wrote or with NumPy built-in functions, was there any performance benefit?

3. Try to use data sharing constructs to push the array to engines in advance, instead of passing them as parameter. Is the code simpler? Did you get any performance benefit?
4. Try to use a gather command to receive results from engines instead of simple return.

3. Advanced Task

The goal of this task is again to approximate the value of π, but this time using Monte-Carlo based approach, which fits parallel processing very well.

The basic principles of this approach are very simple. Consider a rectangle with unit length edge. Now let a circle be enclosed tightly in that rectangle. Since the area of the square is equal to $(2r)^2$, we can find a relation between the area of the circle and that of the square to extract the value of π.



We have the following equations for the area of the square and the circle:
$A_{square} = (2r)^2 = 4r^2$
$A_{circle} = \pi r^2$

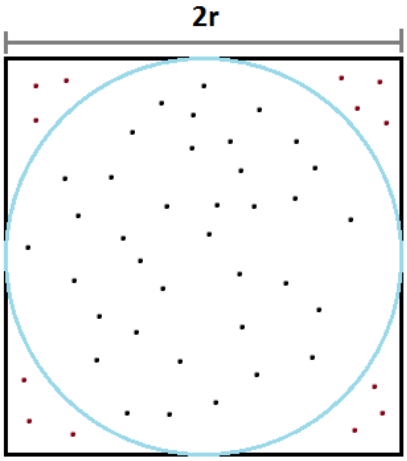
So π can be given exactly as:
$$\pi = 4 \cdot \frac{A_{circle}}{A_{square}}$$

But the setting is that we don't know the areas exactly. All we are given is N random points that lie inside a square of unit length edge and some circle is enclosed within it with a radius $r = 0.5$.

The Monte-Carlo method works as follows:

- Generate N random points to fall within the square:
  - Randomize N times x coordinates from a uniform distribution in the interval [0, 1].
  - Repeat the same for y coordinates.
- Count how many points are enclosed within the circle.
- Then:

$$\pi = 4 \cdot \frac{N_{points-in-circle}}{N}$$



All operations should be performed in the interactive prompt.

1. Open a new terminal to your working node.

2. Start a controller manually and attach engines by executing manual commands, such that the number of engines is equal to the number of CPU cores available on your node minus 1 (8 cores => 7 engines).

3. Open a new terminal, start an IPython interactive prompt.

4. Import the necessary modules and create a client to connect the controller. Verify that the number of reported engines matches your settings.

5. Using NumPy, generate two random arrays of size N, floating point elements, from a uniform distribution in the interval [0, 1]. These arrays will serve as x, y coordinates.

6. Write a function to be running in parallel, call it *my_mc_pypi*. The function should receive as input the following parameters as a packed argument:
   - **x** – The full array containing N random elements for x coordinates
   - **y** – The full array containing N random elements for y coordinates
   - **i_start** – Initial index to start scanning the array from
   - **i_stop** – Final index to stop scanning the array at (not inclusive)

   When called, the function will compute the number of points within that subset that fall inside the circle and return it.

7. Run the computation over the entire array in parallel using the *map* function, so that each engine (task) receives an equal amount of work.

8. Upon return to the caller (interactive prompt), compute and print the approximation to π, also print the error to "exact" pi as in the previous exercise.

9. Perform the above procedure for different values of $N = 10, 100, 1000, 10000, \dots, 10^9$.
   What is the number of generated points to get a satisfying accuracy? (e.g. less than $\epsilon = 10^{-6}$).

Some additional remarks and programming challenges:

1. You may also print the total runtime it took to compute π in parallel.
2. Compare it with a serial version that you wrote, was there any performance benefit? Starting with which value of N?

3. Try to use data sharing constructs to push the arrays to engines in advance, instead of passing them as parameter. Is the code simpler? Did you get any performance benefit?
4. Try to use a gather command to receive results from engines instead of simple return values.

5. For additional performance benefit, you may modify the tasks to accept a single parameter N_{task} that represents the number of points it shall draw internally so no arrays are being passed.
   The task (in the engine) will generate random points by itself and return the same result.

   Be careful to start the random number generator differently for each task, otherwise they will all generate the coordinates.

6. Test your CPU performance and code scalability by creating engines of different count (2, 3, 4, more than the actual CPU core count etc.) and see if the overall computation time improves or degrades.

