Exercise 3 – mpi4py Module

Technical Preliminaries

This exercise requires the use of a standard Python 2.7(.X) or Python 3(.X) installation alongside with mpi4py module and supporting MPI backend (OpenMPI, MPICH etc.).

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. MPI is already configured, use the standard mpiexec command to run programs in parallel.

   To run python 2.7.9 with MPI on 4 processors enter at the command prompt:


```
# mpiexec -n 4 python my_mpi4py_program.py
```

To run python 3.6.0 with MPI on 4 processors enter at the command prompt:

```
# mpiexec -n 4 python3 my_mpi4py_program.py
```


1. A Beginner Task

This preliminary task is very similar to the previous exercise, just to get started.

Using the **mpi4py** module, write a simple python program as follows:

1. Write a basic MPI program to be running in parallel.

2. When executed, the program will print to the screen a message in the form: "Hi, I'm process *ID* from a total of XXX (with *PID*) running on HOST"

   Where ID should be replaced with the rank assigned to the process, XXX with the total number of workers running, PID with the process ID of the running worker and HOST with the hostname the program is running on (may use **socket** module for that).

3. Run the program in parallel using the mpiexec command, for a total of tasks equal to twice the number of CPU cores in your node.

Some remarks:

1. Did any of the programs run on different node?
2. Try to run it on exactly three nodes.

2. Intermediate Task

The goal of this task is to approximate the value of π as in the advanced task of the previous exercise, using Monte-Carlo based approach. Please review the previous exercise for preliminary introduction to the method.

Using the **mpi4py** module, write a python program as follows:

1. Write an MPI program to be running in parallel.

2. Using NumPy, the master generates 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.

3. The master then communicates the arrays to all other workers:
   - **x** – The full array containing N random elements for x coordinates
   - **y** – The full array containing N random elements for y coordinates

4. Upon receiving the arrays, all processes (including the master) will compute the number of points within that subset that fall inside the circle and return it to master via gather operation.
   Make sure that work is evenly distributed between workers (they can infer the indices to scan in the arrays quite easily).

5. After completing the gather, the master will compute and print the approximation to π, and also print the error to "exact" pi as in the previous exercise.

6. Perform the above procedure for different values of $N = 10, 100, 1000, 10000, \dots, 10^9$.

<u>Some additional remarks and programming challenges:</u>

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

3. Try to efficiently pass data to workers such that each receives the exact number of items to process and not the entire arrays. Did you get any performance benefit?

4. For additional performance benefit, you may modify the program so that arrays are not passed at all, but only a single parameter N_{task} that represents the number of points it shall draw internally.

The program will generate random points by itself and return the same result via a gather operation.

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

5. 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.
   What happens when you go out-of-core? (outside your compute node)

3. Advanced Task

The goal of this task is to compute the famous N-body simulation in physics (astrophysics, electrostatics etc.), which fits parallel processing very well.
This simulation is used to track the motion of 3 or more particles in the presence of other forces and particles in the universe.

For simplicity purposes, we will use the Particle-Particle solution scheme. The basic concept behind it is to freeze the world (so quantities like acceleration and velocity remain constant), then for each particle compute the net forces acting on it and advance its location to next time step. This is performed for all particles before moving to the next time step.
You can get a feeling how intensive it can get once the simulation includes billions of interacting particles (thus, we need supercomputers).

From all possible physical problems, we shall address the one related to astrophysics/cosmology and involving simple gravitational forces.

We will use the SI (metric) system for physical units. Our tiny world for this setting will be 2D with axes in the interval [-100, 100] for each direction. Particles' location is drawn from a uniform random distribution in that interval. Velocities will be in the uniform interval [-1, 1]. Masses are expressed in kilograms and will be in the uniform interval [10000, 100000].

NOTE: You may modify the physical settings at your preference for something else that will work.

Using the **mpi4py** module, write a python program as follows:

1. Write an MPI program to be running in parallel.

2. Define the final time t_{final} (orders of seconds, up to 100 [s]) that the simulation should proceed and also the time-step dt (check 0.1 or 0.01) for each computational step of the simulation. These values will be defined as constants in the program code.

3. Using NumPy, the master generates the following initial information:
   - x_0 – Random x coordinates of N particles in space
   - y_0 – Random y coordinates of N particles in space
   - $v_{0,x}$ – Random velocity in x direction for each particle
   - $v_{0,y}$ – Random velocity in y direction for each particle
   - m – Random mass for each particle

4. The master then communicates the arrays to all other workers.

5. For each time-step (until t_{final}) perform the following:
    1. For each particle (i):
        i. Compute the net force acting on it in each direction using the superposition principle:

$$F_{i,x} = Gm_i \cdot \sum_{i \neq j}^{N} \frac{m_j}{r_{ij}^3} \cdot \left(x_j - x_i\right)$$

$$F_{i,y} = Gm_i \cdot \sum_{i \neq j}^{N} \frac{m_j}{r_{ij}^3} \cdot \left(y_j - y_i\right)$$

And $G = 6.67 \cdot 10^{-11}$, $r_{ij} = \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2}$

        ii. Compute the acceleration using the recent net force and Newton's second law of motion:

$$a_{i,x}(t) = \frac{F_{i,x}}{m_i}$$

$$a_{i,y}(t) = \frac{F_{i,y}}{m_i}$$

        iii. Update the new velocity using the recent acceleration, we assume that acceleration remains constant within this update process (time interval):

$$v_{i,x}(t + dt) = a_{i,x}(t) \cdot dt + v_{i,x}(t)$$

$$v_{i,y}(t + dt) = a_{i,y}(t) \cdot dt + v_{i,y}(t)$$

        iv. Update the new location using the recent velocity, we assume that velocity remains constant within this update process (time interval):

$$x(t + dt) = v_{i,x}(t) \cdot dt + x(t)$$

$$y(t + dt) = v_{i,y}(t) \cdot dt + y(t)$$

6. All workers (including master) participate in the computation of each time-step. Make sure that work is evenly distributed between workers, where for simplicity you may assume that each worker can process at least one particle.

7. When finishing a time-step, all worker must notify the master for the new physical variables of the particles and shall update all workers with new information as well.

8. After finishing the simulation, the master will save the final results to a file so you can review them.

9. Perform the above procedure for different values of $N = 10 \rightarrow 500$. Don't go beyond this as your simulation may run for too long.

Some additional remarks and programming challenges:

1. Try to save particles' location and other physical quantities to a file.
2. Use that information to visualize the simulation for each time-step.
3. Try to use advanced communication methods such as broadcasting and reduce operations, see if it improves runtime.

