QUESTIONS

Q1. Matrix-vector multiplication in parallel.

The goal of this question is to write a parallel code that multplies a 4×4 matrix with a vector v with 4 MPI processes using the so called ”Checkerboard block decomposition”. The code should use the numpy module.

Consider the matrix A and the vector v,

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad v = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{1}$$

For convenience, we define the block matrices b_0, b_1, b_2 and b_3 to be:

$$b_0 = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, b_1 = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}, b_2 = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}, b_3 = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$$

and the vectors:

$$x = \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}, y = \begin{pmatrix} v_2 \\ v_3 \end{pmatrix}$$

.

So that,

$$Av = \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_0 x + b_1 y \\ b_2 x + b_3 y \end{pmatrix} \tag{2}$$

In order to perform the matrix-vector multplication Av in parallel, each process must receive a 2×2 block matrix b_i and the vector x or y. The 4 processes can then compute the following 2-component vectors:

- process 0: computes $u_0 = b_0 x$
- process 1: computes $u_1 = b_1 y$
- process 2: computes $u_2 = b_2 x$
- process 3: computes $u_3 = b_3 y$

Once each process has performed its calculation the results are gathered on process 0 and combined to obtain Av.

(a) Write a function that returns a random vector of length n.

(b) Write a function that return a random square matrix of size $n \times n$.

(c) Use the two functions developed in (a) and (b) with $n = 4$ to generate a random matrix A of size 4×4 and a 4-dimensional vector.

(d) Write a function that takes in input a 4×4 matrix A, and returns a list l of the 2×2 block matrices ($l = [b_0, b_1, b_2, b_3]$).

(e) Write a function that takes in input a four dimensional vector v, and returns a list l of four two dimensional vectors ($l = [x, y, x, y]$).

(f) Use the functions defined in (d) and (e) on A and v, and scatter the result so that each process get the appropriate matrix block matrix b_i and the appropriate vector x or y.

(g) Compute the local matrix vector multiplication u_i (a two dimensional vector multiplied by a 2×2 matrix).

(h) Gather the u_1, u_2, u_3 and u_4 on process 0, and build up the vector Av using Eq. 2. Check using a direct calculation of the product Av that your result is correct.

Q2. Simulating an ideal gas of particles.

Consider a two dimensional box of side L, filled with N particles that do not interact with each other. The particles have the same mass and their size is negligible compared to the volume of the box. The particles only interact with the boundary of the box (reflecting walls).

You can find on the DLE a serial code (`gas_serial.py`) that simulates such an ideal gas. Given an initial position and velocity distribution of the particles, the code evolves the system following Newton second's law for a number N_{iter} of time steps dt.

(a) Describe the class Particles. How is initialised ? What is the role of each method?

(b) What are the distributions of the initial positions and velocities ?

(c) Parallelise the code so that each process simulates the motion of N particles. The part of the code generating the plot and the animation should only be run by process 0.

(d) Modify the code to compute numerically the averaged kinetic energy K per particle. Since the energy is conserved, you can compute the average kinetic energy of a particles for the initial distribution of velocities.

(e) The collision of a particle with the box generates a force on the wall. Using classical mechanics, the force applied by one particle on the wall located at $x = L$ during an interval dt reads $f_x = 2mv_x/dt$ where v_x is the velocity of the particle in the x-direction (the component orthogonal to the boundary). The force F_x exerted by N particles colliding with the boundary in a dt interval is the sum of the individual forces. The pressure is then defined to be the force per unit of length and thus read $P = F_x/L$.
Compute the average pressure during an interval dt over N_{iter} steps for each process and globally.

(f) Defining $V = L^2$ (the volume of the box) and $N_{\text{tot}} = pN$(the total number of particles simulated), plot PV and the averaged total kinetic energy $N_{\text{tot}}K$ for a few volumes keeping the number of particles fixed. In the limit of a very large number of particles, the two quantities are equal. This is a famous result, which relates macroscopic quantities (the pressure and the volume) to the average energy of the particles (a microscopic property)! It can be derived analytically.

(g) Suppose now that particles interaction are switched on, briefly discuss on paper how the code would have to be modified, and what would be the main challenges.

Q3. Performances of a sorting algorithm (parts of this exercise require to use the cluster in room 205).
In computer science a sorting algoritm is an algorithm that puts element of a vector in a certain order. Efficient sorting is important for many applications. In this exercise you will investigate some aspects of the parallelisation of a sorting problem.
Consider a vector of n integer to be sorted. In parallel, the vector is split in subvectors each of them being sent to one of the p process (scatter operation). Each process then sort its own subvector of length n/p (n has to be a multiple of p). The sorting of the subvector itself, is a problem in its own, and we use here the built-in numpy sorting algorithm which (by default the so called the 'quicksort' algorithm).

Once the subvector are sorted, the problem is to merge the sorted subvectors and sort them to obtain the final sorted vector. This is the so-called merging operation designed to merge two sorted vectors. Clearly, one possibility is to do a gather operation and then merge one after the other the sorted vectors on the process 0. This is however an inefficient strategy, since process 0 only will have to merge the subvectors while all the other process will be idle. Several implementation of the merge function are implemented.

The parallel code that you are provided on the DLE (`merge_sort_mpi.py`) sorts a vector of random integer using MPI on an any number of processes. The code runs on any number of processes, but it is meant to be used only on an even number of processes.

(a) Add comments to the parallel code. In particular, the types of the main variables should be added in a comment when declared (for instance : "a vector of floats of size ...").

(b) Explain, by providing an example on paper, how the merging of two vectors of different length is performed in the `merge` function. The function can be used by using `merge_flag = 0`. In a practice a more efficient alternative implementation function `merge_alt` is used by using `merge_flag = 1`.

(c) By reading carefully the code and running it with $p = 4$ or 8, explain how the communications and the merging of the sorted arrays are organised. You can provide a sketch to illustrate the parallelisation.

(d) Supplement the MPI code by implementing the gathering all the sorted arrays back to process 0 and then merging them. Compare the cost with the original implementation to solve a given problem on a 8 processes. Make sure that the vector of integers takes at least 10 megabytes of memory.

(e) Perform a strong and weak scaling efficiencies analysis of the original MPI code on an even number of processes (up to 16 processes, the number of processors in room 205). Provide plots of the efficencies. What can you say about the performance of the code ? Make sure that the vector of integers takes at least 10 megabytes of memory.

END OF QUESTIONS