Tutorial Session: Coding Exercises on MPI - Part II

Question 1: Distributed reduction

We will use MPI to calculate the following sum:

$$x_{\text{tot}} = \sum_{n=1}^{N} n = 1 + 2 + 3 + \ldots + (N-1) + N \tag{1}$$

This sum can be computed by an analytic formula, which we will use to validate our MPI implementation:

$$x_{\text{tot}} = \sum_{n=1}^{N} n = \frac{N(N+1)}{2} \tag{2}$$

a) Initialize and finalize MPI, and get rank and size of communicator.

b) First, we perform the reduction using MPI blocking collectives, in function `reduce_mpi`. Note that each rank computes only a part of the sum. Distribute the workload reasonably, in order to guarantee load balancing. Specifically, each rank should calculate a partial sum:

$$\text{sum}_{\text{rank}} = N_{\text{start}} + (N_{\text{start}} + 1) + \ldots + N_{\text{end}} \tag{3}$$

c) Then, we implement our own reduction in function `reduce_manual`. This can be done in a tree-like way as depicted in Fig. 1. Our task is to implement this scheme for the special case that the number of ranks is a power of 2.

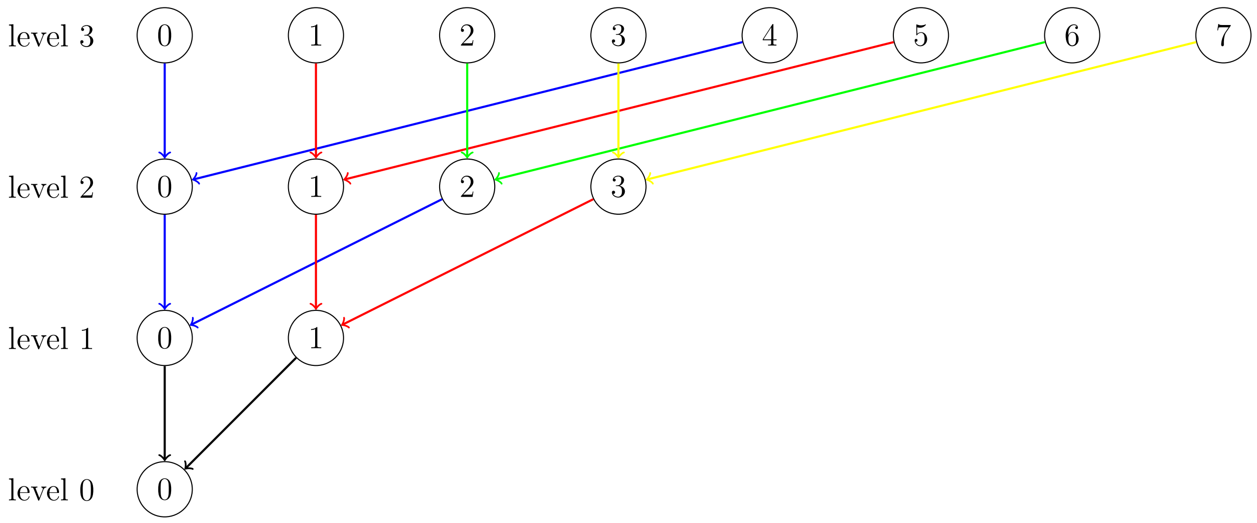

Figure 1: The communication pattern of a tree-like reduction. Each circle represents a rank, the number inside is the rank ID. Communication takes place along the arrows.

d) What is the advantage of the tree-like scheme compared to the naive reduction?

Consider that in the naive approach, every rank sends its elements directly to the master. The master then reduces all obtained elements by repeatedly applying the operation, in our case the sum. The two main advantages of the tree-like reduction are:

- More ranks communicate with each other. Depending on the network topology, this means that there is potentially more bandwidth available.
- The reduction operation can be performed by many processes in parallel, leading to better load balance and faster execution of the reduction. In the naive version, every process has to wait until the root of the reduction has performed the reduction operation on all items received by another rank. Also, only one process can send at each time, leaving the others idle and wasting an opportunity to do something useful.

Question 2: Roll-up of a vortex line

We want to simulate the evolution of a two-dimensional vorticity sheet, as a simple model for the wake of an airplane (Figure 2). We use N particles moving with time t, each particle i is located at $(x_i(t), y_i(t))$ and carries a constant value Γ_i (circulation).

Initially, particles are uniformly placed within the domain $x \in [-0.5, 0.5]$, and have circulation $\Gamma_i = \frac{1}{N}\frac{4x_i}{\sqrt{1-4x_i^2}}$. This configuration defines the Initial Conditions of our system, implemented inside function `initialConditions`.

According to our simple wake model, the velocity field is computed as a function of the particles' positions and circulation:

$$u(x_j, y_j, t) = \sum_{i=0}^{N-1} \frac{\Gamma_i}{2\pi} \frac{-\left[y_j(t) - y_i(t)\right]}{\varepsilon^2 + \left[x_j(t) - x_i(t)\right]^2 + \left[y_j(t) - y_i(t)\right]^2},$$

$$v(x_j, y_j, t) = \sum_{i=0}^{N-1} \frac{\Gamma_i}{2\pi} \frac{\left[x_j(t) - x_i(t)\right]}{\varepsilon^2 + \left[x_j(t) - x_i(t)\right]^2 + \left[y_j(t) - y_i(t)\right]^2}.$$

This is an approximate solution of Euler equations and corresponds to the vorticity field

$$\omega_j = \frac{\partial v(x_j, y_j, t)}{\partial x_j} - \frac{\partial u(x_j, y_j, t)}{\partial y_j} = \sum_{i=0}^{N-1} \Gamma_i \delta_\epsilon\big(x_j(t) - x_i(t), y_j(t) - y_i(t)\big),$$

where $\delta_\epsilon(x, y) = \frac{1}{\pi}\frac{\epsilon^2}{(\epsilon^2 + x^2 + y^2)^2}$ approximates a Dirac-delta function for a small ϵ. Over time, particles move depending on the velocity at their location,

$$\frac{dx_i(t)}{dt} = u(x_i(t), y_i(t), t), \qquad \frac{dy_i(t)}{dt} = v(x_i(t), y_i(t), t).$$

a) Implement the interaction function `computeVelocities` in the file `vortex-line/skeleton_code/serial.cpp`.

The results may be checked by visualizing the generated csv files with a visualization software. We provide a plotting tool inside `vortex-line/solution_code/plot/plot.py` that plots the particles' position at different time snapshots.

b) Parallelize your code using MPI by filling in the TODOs in `vortex-line/skeleton_code/mpi.cpp`. Each MPI rank must contain an equal number of particles. The parallelization of function `computeVelocities` can be done using a multi-pass communication, as described in the table:

		process p				
		0	1	$\cdots$	P-2	P-1
pass q	0	D_0	D_1	$\cdots$	D_{P-2}	D_{P-1}
	1	D_{P-1}	D_0	$\cdots$	D_{P-3}	D_{P-2}
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	P-1	D_1	D_2	$\cdots$	D_{P-1}	D_0

The data D_p necessary to compute the velocities (x_i, y_i and Γ_i) must be communicated to every rank in a cyclic manner until every rank has computed the interactions between its own particles with every particles in the simulation. Parallelize the function `dumpToCsv` by gathering the data on the root.

c) Use non-blocking MPI routines to overlap the communication with computation. Write your solution in `vortex-line/skeleton_code/mpi_non_blocking.cpp`.

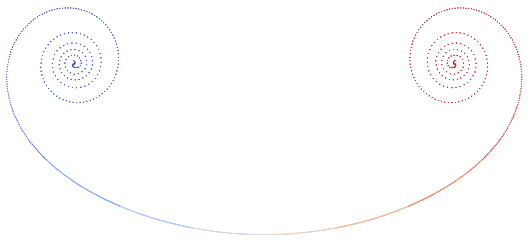

Figure 2: Left: Wake of an airplane visualized by condensation behind the engines. The vorticity sheet is generated at the trailing edge of the wings. Right: Vortex sheet at $t = 1$ from the simulation.