P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2019

Set 5 - Particle Methods and MPI

Issued: November 22, 2019
Hand in (optional): December 6, 2019 8:00am

Question 1: Roll-up of a vortex line

In this exercise you will simulate the evolution of a two-dimensional vorticity sheet. Vorticity sheet describes a fluid flow with a discontinuity of the velocity field $\mathbf{u}(\mathbf{x}, t)$ along a surface (the sheet), such as in slippage of one layer of fluid over another. As the name suggests, the vorticity $\omega(\mathbf{x}, t) = \nabla_{\mathbf{x}} \times \mathbf{u}(\mathbf{x}, t)$ of such flow is non-zero only over the sheet, or, because we consider only a two-dimensional cross-section of the field, along a line.
We aim to carry out the simulation by discretizing the flow with N point vortices distributed along the vortex line, such that the i-th particle is located in $\mathbf{x}_i(t)$, has volume $\text{Vol}_i = 1/N$, and carries circulation $\Gamma_i = \int_{\text{Vol}_i} \omega_i \mathrm{d}\mathbf{x}$. The total vorticity field can be reconstructed as:

$$\omega(\mathbf{x}, t) = \sum_{i=0}^{N-1} \Gamma_i \delta(\mathbf{x} - \mathbf{x}_i), \tag{1}$$

The velocity field $\mathbf{u}(\mathbf{x}, t) = \{u_x(\mathbf{x}, t), u_y(\mathbf{x}, t)\}$ described by this distribution of point vortices is given by:

$$u_x(\mathbf{x}, t) = \sum_{i=0}^{N} \frac{\Gamma_i}{2\pi} \frac{-[y - y_i(t)]}{[x - x_i(t)]^2 + [y - y_i(t)]^2}, \tag{2}$$

$$u_y(\mathbf{x}, t) = \sum_{i=0}^{N} \frac{\Gamma_i}{2\pi} \frac{[x - x_i(t)]}{[x - x_i(t)]^2 + [y - y_i(t)]^2}. \tag{3}$$

The time-evolution of the vortex line is obtained by advecting the particles along the velocity field:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}_i = \mathbf{u}(\mathbf{x}, t)|_{\mathbf{x}=\mathbf{x}_i} \tag{4}$$

Therefore, for each time step, in order to update the position of the i-th particle, one has to compute the velocity field at location $\mathbf{x}_i(t)$, which involves calculating its interaction with all other N particles (**excluding self-interaction**). Because we assume that the circulation Γ_i carried by each particle is constant in time, we neglect dissipation.

a) (**3 points**) In order to gain some intuition about the set of equations we just described, consider two vortices with circulations Γ_1 and Γ_2 initially located at $bx_1(0) = (\Delta/2, 0)$ and $\mathbf{x}_2(0) = (-\Delta/2, 0)$. Use equations (2) and (3) to describe the motion of the vortices $\dot{\mathbf{x}}_1(t)$ and $\dot{\mathbf{x}}_2(t)$ for two cases: (i) $\Gamma_1 = \Gamma_2 = \Gamma$, (ii) $\Gamma_1 = -\Gamma_2 = \Gamma$.

Figure 1: Wake of an airplane visualized by condensation behind the engines. The vorticity sheet is generated at the trailing edge of the wings.

The remaining part of the exercise is to implement a distributed simulation of the roll-up of a vortex line, discretized over an arbitrary number of point vortices. The solution should resemble a cross-section of the wake an airplane (see Figure 1).

Initially, N vortex particles are arranged uniformly along the x-axis in $(-\frac{1}{2}, \frac{1}{2})$:

$$x_i = -\frac{1}{2} + \frac{i + 1/2}{N}, \quad i = 0, ..., N-1 \tag{5}$$

and have circulation $\Gamma_i = \text{Vol}_i \, \frac{4x}{\sqrt{1-4x^2}}$ with $\text{Vol}_i = 1/N$.

The skeleton code for this exercise consists of the following:

- `ArrayOfParticles.h` : defines a class that stores the positions $\mathbf{x}_i$, velocities $\mathbf{u}_i$ and circulations Γ_i of a set of particles. Should not be modified.
- `Profiler.h` : defines a class that helps time the various parts of your solver. Should not be modified.
- `SerialParticlesIterator.h` : defines methods to perform operations on the array of particles. Such as: (i) initialize $\mathbf{x}_i$ and Γ_i; (ii) reset $\mathbf{u}_i$ to zero; (iii) adding to the velocities $\mathbf{u}_i$ of a set of "target" particles the contribution due to the circulation carried by a set of "source" particles; (iv) advecting the particles' positions with the updated velocities.
- `main.cpp` : Defines the main simulation loop for a serial code.

The code can be compiled and run on a compute node of Euler by loading the modules `module load new gcc/6.3.0`.

b) (**3 points**) Write `SerialParticlesIterator::compute_interaction`. For each particle in the array "`targets`", add (using equations (2) and (3)) the contributions to its velocity due to the particles in the array "`sources`". This function completes the serial solver and it should now carry out a correct simulation.

Note that, for the serial version, sources and targets are the same particles and you need to neglect self-interaction. However, for the distributed version in the next steps, the two arrays may hold different sets of particles.

c) (**3 points**) Duplicate your `SerialParticlesIterator.h` (you will need a serial version for later) and implement a version parallelized with OpenMP threads. All the functions in this file (and exclusively those) need be changed for this step.

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

Table 1: Multi-pass approach to compute N^2 interactions.

d) (**3 points**) Tweak the binding/scheduling options of OpenMP and produce the strong and weak scaling plots starting from $N_1 = 3600$ for 1 OpenMP thread with at least 3 points per plot. Note: the solver's computational complexity is defined by its most expensive component.

For the next steps, you will have to duplicate your `main.cpp` and prepare a distributed-memory version with MPI (and **not** using OpenMP):

e) (**3 points**) Distribute the particles across P processes such that the first rank holds the first N/P covering the first length $1/P$ of the simulation domain, and so on.

f) (**3 points**) Modify the I/O lines of the solver (look for `fwrite`) to use collective write operations (**hint** you can keep it simple by using `MPI_File_write_ordered`).

g) (**5 points**) Evaluate the velocity of the particles using a multi-pass approach where the P processes exchange particles P times. Let D_p be the N/P particles assigned to process p. Each process must store in its memory an additional array of N/P particles (let's call it D_q). At every pass, each processor sends its particles D_p to one other rank, receives the particles from another rank and stores them in the fields of D_q. Once the exchange is finished, each process computes the interactions between D_p and D_q. During the first pass, each process computes the interaction among its own D_p and therefore no interaction is needed.
Table 1 illustrates a possible data arrangement.

h) (**2 points**) At each step, compute the total circulation carried by the particles.

i) (**3 points**) Modify the multi-pass approach in order to overlap communication and computation (**hint** you will need an additional temporary storage array of particles).

j) (**3 points**) Make weak and strong scaling plots for the distributed solver starting from $N_1 = 3600$ for $P = 1$ with at least 3 points per plot. Comment on your findings.

k) (Optional **3 points**) Use the configuration files to check that your distributed solver produce the same output as the serial and multi-threaded versions. Run your code with $10\,000$ particles and plot the particle positions at times $t = 0.5$, $t = 1.0$ and $t = 2.0$.

l) (Optional **3 points**) Combine your MPI and OpenMP parallelization approaches. **Note**: hybrid jobs are not fully supported on Euler and therefore it would take some fighting with modules and environment variables for this code to scale. You can ignore these challenges for this step: work only on the code.