HPC for mathematicians

Workshop 3 (week 4)
Point-to-Point Communications

February 1, 2021

Exercise 1

In this task, you get to use '`MPI_Send`' and '`MPI_Recv`' in a very simple example, where you test exchanges of a few messages. Consider **c** to be a vector of size 7 and A a 3×3 matrix. Hire two processors and investigate the following exchanges (one of the processors is sending a message and the other receives the same message). You should include print statements before '`MPI_Send`' and after '`MPI_Recv`' to see what is in **c** and A.

1. Fill A with zeros in process 1 and set $\mathbf{c} = [1\ 2\ 3\ 4\ 5\ 6\ 7]$ in process 0. Send **c** from process 0 and received the message by process 1 and store it in A with the 'count = 7'. How does A look like after receiving the message?
2. In process 0 fill A as below
$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \qquad [1.1]$$
and in process 1 fill **c** with zeros. Then send a message from process 0 to 1 using A as the send buffer and **c** as the receive buffer. Keep 'count = 7'.
3. Repeat the previous example but change A to
$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \qquad [1.2]$$

Optional: use dynamics memory allocation to define **c** only for one process and A for the other process.

Exercise 2

The aim of this task is to get familiar with point-to-point communication in a simple example. Two processes collaborate together to calculate $N!$. First, the process rank 0 calculates 1! and then passes it to the process rank 1. Once rank 1 receives the message, calculates $2! = 1! \times 2$ and returns it back to rank 0. This back-and-forth continues until $N!$ is calculated. Before sending their message, each process should print:

“I am process X and going to send Y"

where **X** is the rank of sender and **Y** is the updated value that is going to be sent to the receiver. You can view this as a ping-pong game between two processes.

Exercise 3

In this exercise, we use halo-swapping technique to solve 1D heat equation in parallel. In so doing, we need to employ point-to-point communication functions of the MPI library.

Let's consider the following heat equation:

$$u_t = u_{xx} \quad \text{for } x \in (0,1), t \in (0,T), \tag{3.1}$$

subject to an initial condition

$$u(x, t=0) = \sin(2\pi x) + 2\sin(5\pi x) + 3\sin(20\pi x), \tag{3.2}$$

and homogeneous Dirichlet boundary conditions

$$u(x=0,t) = u(x=1,t) = 0. \tag{3.3}$$

Applying the method of lines, and discretising over the t-dimension with a forward Euler discretisation, leads to

$$\frac{U_m^{n+1} - U_m^n}{\Delta t} = \frac{U_{m-1}^n - 2U_m^n + U_{m+1}^n}{(\Delta x)^2} \quad \text{for } m \in \{1,\dots,M-1\}, n \in \{0,\dots,N-1\}, \tag{3.4}$$

with

$$U_m^0 = \sin(2\pi x_m) + 2\sin(5\pi x_m) + 3\sin(20\pi x_m) \quad \text{for } m \in \{1,\dots,M-1\}, \tag{3.5a}$$

$$U_0^n = U_M^n = 0 \quad \text{for } n \in \{0,\dots,N\}, \tag{3.5b}$$

and where U_m^n is the fully discrete approximation for $U(x = x_m = m\Delta x, t = t_n = n\Delta t)$, with $\Delta x = 1/M$, $\Delta t = T/N$ for some positive integers M, N. In this discretisation, to update any point at step $n+1$, we need three points from previous time-step: U_m^n, U_{m-1}^n and U_{m+1}^n, which is demonstrated in the stencil of figure 1.

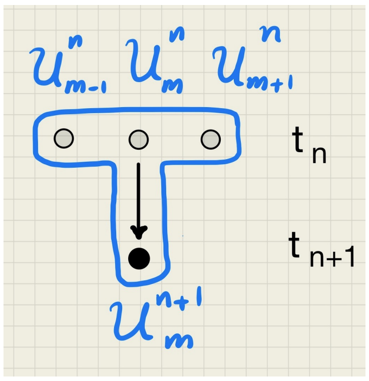

Figure 1: Stencil of central differencing in space and forward Euler in time for heat equation.

First, write a serial code that solves this equation for $T = 0.1$ using the discretisation in (3.4). The exact solution to this problem is

$$u(x,t) = e^{-4\pi^2 t}\sin(2\pi x) + 2e^{-25\pi^2 t}\sin(5\pi x) + 3e^{-400\pi^2 t}\sin(20\pi x). \tag{3.6}$$

Use this solution to verify the answer of your serial code. Note that Forward Euler is numerically unstable for $\Delta t/(\Delta x)^2 > 0.5$; hence, you need to make your Δt is small enough such that $\Delta t/(\Delta x)^2$ is less than 0.5. I have included a high-wavenumber term in the initial condition so you need to choose a small Δx as well to have an accurate solution at early times.

Once you are confident with your serial code, you can parallelise the computation using MPI. A simple approach is to divide the physical space (x-dimension) into sub-intervals and assign each sub-interval to a process to update the values of U_i within their sub-interval. Considering the stencil shown in figure 1, we don't have enough information for the grid points at two ends of interval unless they are boundary points. To make this clear, let's use the notation $U_{p,\,j}$, where p is the rank of process and j is a spatial index varying from 1 to J (J being the number of grid points in each interval). For example, the required information at step n is not available to derive $U^{n+1}_{p,\,0}$ according to equation (3.4).

To solve this issue, we are using a technique called 'halo-swapping'. First, we overlap the sub-intervals by two points as shown in the figure below. For example, the last two points of interval 0 corresponds to the first two points of interval 1.

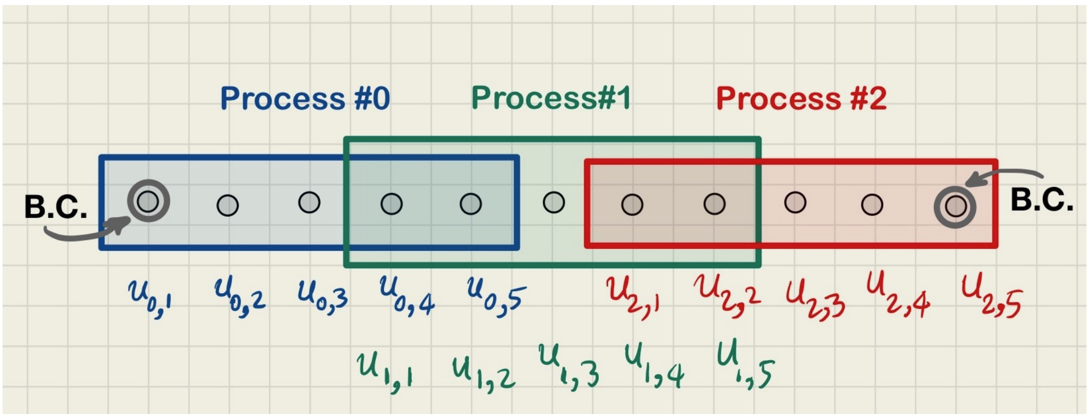
Process #0
Process#1
Process #2
B.C.
B.C.
$u_{0,1}$ $u_{0,2}$ $u_{0,3}$ $u_{0,4}$ $u_{0,5}$
$u_{2,1}$ $u_{2,2}$ $u_{2,3}$ $u_{2,4}$ $u_{2,5}$
$u_{1,1}$ $u_{1,2}$ $u_{1,3}$ $u_{1,4}$ $u_{1,5}$

Figure 2: Overlapped intervals for halo swapping.

Now each interval is able to update all their middle points using the forward Euler scheme in (3.4) (the very first and last points cannot be updated). This is shown in the figure below, where the updated points in each interval are filled with black.

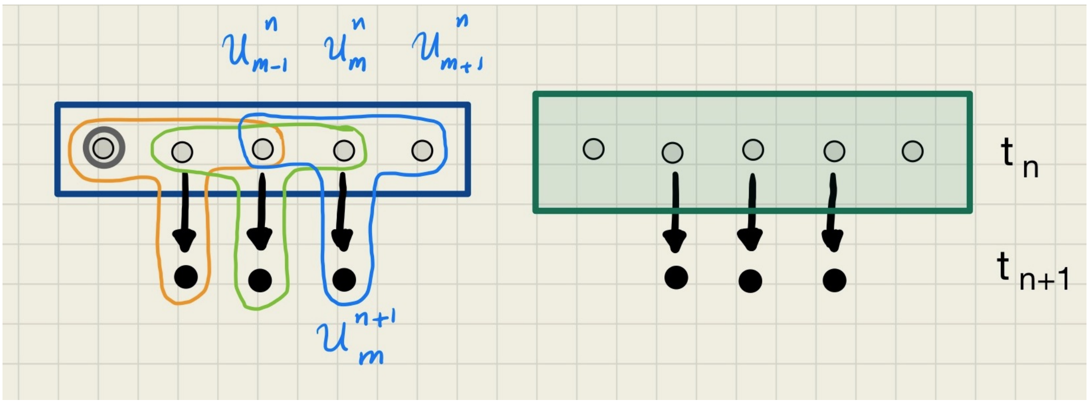

Figure 3: Updating the middle points in each time interval following (3.4).

Now if we put all the intervals together, we see that missing information in each interval can be obtained from the neighbouring intervals as shown in figure (4). For example, the end point

$U_{p,J}$ can be obtained from $U_{p+1,2}$ as it is a middle point in the interval $p+1$. This can be done with a proper send and receive functions in MPI.

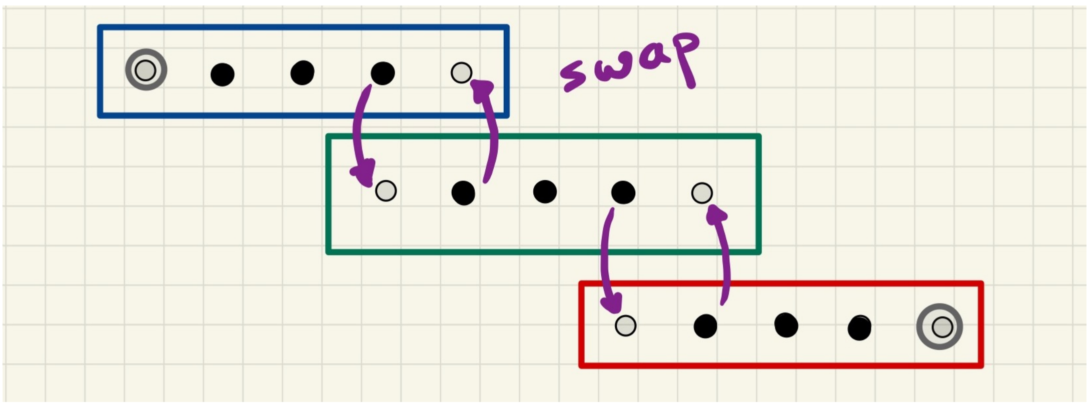

Figure 4: All the intervals together. The filled points are updated for each interval. (3.4).

After writing the parallel code using the technique above, verify its validity by comparing it with your serial code. Technically you should get the same results from both code. Then, try to run it using different number of processors. Report the computation time for each case.

For this problem, you do not need to take care of general case of M and the the total number of processor (size). You can assume that the user is going to pick the correct values. To pick the right number of processes and grid size such that all the processes get equal intervals the following relation should hold

$$M = size \cdot (J - 2) + 2, \qquad [3.7]$$

where M is the number of grid points, $size$ the number of processes and J the number of grid points given to each process (including overlaps). Hence, M and $size$ should be chosen such that (M-2)/size is an INTEGER number bigger or equal to 3 (preferably much larger so overhead communication would not take over parallel computing).