Problem 2 - MPI Domain Decomposition [60%]

In this problem, we wish to model the 2D wave equation:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

where u can be thought of as the wave "height" and the above equation defines its evolution in time and space. We model this problem on the domain $(x, y, t) \in [0, 1] \times [0, 1] \times [0, T]$ with the boundary conditions:

$$u(x,y,0) = u^0(x,y) \qquad \frac{\partial}{\partial t}u(x,y,0) = 0$$
$$\frac{\partial}{\partial x}u(0,y,t) = 0 \qquad \frac{\partial}{\partial x}u(1,y,t) = 0 \qquad \frac{\partial}{\partial y}u(x,0,t) = 0 \qquad \frac{\partial}{\partial y}u(x,1,t) = 0$$

Define

$$u^n_{i,j} = u((i-1)\Delta x, (j-1)\Delta y, n\Delta t)$$

for $i = 1, \ldots, Nx$ and $j = 1, \ldots, Ny$. Additionally, $\Delta x = 1/(Nx - 1)$ and $\Delta y = 1/(Ny - 1)$ are the grid spacings so that $u^0_{1,1} = u(0, 0, 0)$ and $u^0_{Nx,Ny} = u(1, 1, 0)$.

We can write the continuous PDE in terms of our grid of values using second-order central difference approximations:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^n_{i-1,j} - 2u^n_{i,j} + u^n_{i+1,j}}{\Delta x^2}$$
$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u^n_{i,j-1} - 2u^n_{i,j} + u^n_{i,j+1}}{\Delta y^2}$$
$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u^{n-1}_{i,j} - 2u^n_{i,j} + u^{n+1}_{i,j}}{\Delta t^2}$$

Substituting these into the wave equation, letting $\Delta x = \Delta y$ (and $Nx = Ny$), and solving for $u^{n+1}_{i,j}$, we derive the updating scheme for our grid:

$$u^{n+1}_{i,j} = (2 - 4\frac{\Delta t^2}{\Delta x^2})u^n_{i,j} - u^{n-1}_{i,j} + \frac{\Delta t^2}{\Delta x^2}(u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1}) \tag{1}$$

Thus, if we assume we know $u^n_{i,j}$ and $u^{n-1}_{i,j}$ for all i, j, then we can compute the next step in time, $u^{n+1}_{i,j}$, for each i, j.

This is called a computational stencil and defines the dependencies of the computation. In order to compute the value $u^{n+1}_{i,j}$, we need the values $u^n_{i-1,j}, u^n_{i+1,j}, u^n_{i,j-1}, u^n_{i,j+1}, u^n_{i,j}$, and $u^{n-1}_{i,j}$. This is depicted in Figure 1.

If we were to use this stencil to compute $u^n_{i,j}$, we would also require points that we don't represent and are outside of the problem's domain, for example $u^n_{0,j}$ and $u^n_{i,Ny+1}$. Typically,

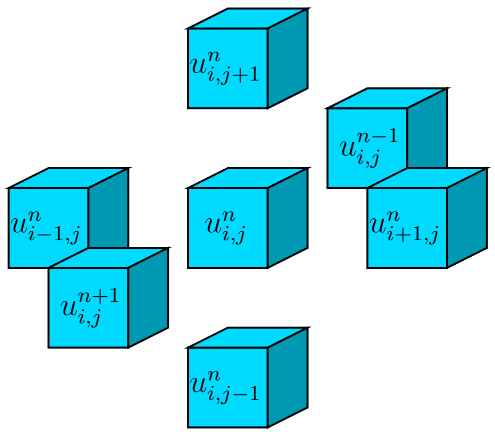

Figure 1: Spacial depiction of the computational stencil.

we would have to test if we are currently updating the boundary to make sure we don't go off the edge of the data. Alternatively, we can pad the domain for dummy values of these "outside" grid locations so that we don't have to modify the stencil on the boundaries. Then, we need only force these padded grid points to have the correct value. This is called the method of *ghost points*.

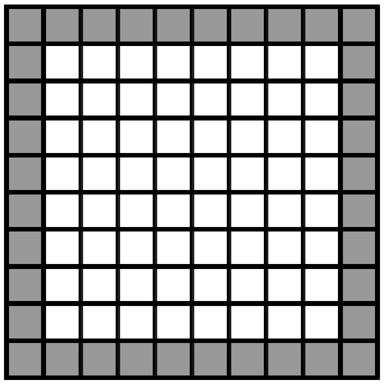

Figure 2: Ghost points surrounding our the grid so that we can update all of the interior grid points with the computational stencil.

To determine what values the ghost points should have, we note that the central difference approximation of the boundary condition

$$0 = \frac{\partial u}{\partial x}(0, y, t) \approx \frac{u^n_{0,j} - u^n_{2,j}}{2\Delta x}$$

implies that $u^n_{0,j} = u^n_{2,j}$. Similar results hold for the other ghost points:

$$u^n_{0,j} = u^n_{2,j} \qquad u^n_{Nx+1,j} = u^n_{Nx-1,j} \qquad u^n_{i,0} = u^n_{i,2} \qquad u^n_{i,Ny+1} = u^n_{2,Ny-1} \tag{2}$$

Therefore, after computing all the interior points for step $n+1$, we can set the ghost values appropriately for computing step $n+2$.

Each iteration then follows the steps

- Compute the interior grid points with the computational stencil for step $n+1$. That is, compute $u_{i,j}^{n+1}$ for all i, j using u^n and u^{n-1} in Equation (1).
- Set the ghost points for u^{n+1} using Equation (2).
- Set $u^{n-1} \leftarrow u^n$ and $u^n \leftarrow u^{n+1}$ and continue to the next step $n \leftarrow n+1$.

The serial version is implemented for your inspection in `P2_serial.py`.

Domain Decomposition

To parallelize this computation, we partition the domain among the processes, giving each process responsibility for a small piece of the domain. In this problem, we will use a rectangular grid decomposition with `Px` and `Py` processes in the x- and y-directions, respectively.

Similar to the serial program, in order to compute u^{n+1} each process will require data from grid points that it is not responsible to update. If *each* process uses the ghost point method – it collects and uses information on its border but does not compute the update stencil at those points – draw what needs to be communicated between processes at each iteration with an image similar to Figure 2.

Implementation

Implement a parallel version of `P2_serial.py` using the rectangular grid domain decomposition above.

You may use any communicator strategy that you like. For example:

- Using the `COMM_WORLD` throughout.
- Using `Split` to create your own communicator(s) as we discussed in class.
- Using `Cartcomm` – a specific communicator designed for Cartesian topologies like this.

You may assume that `Nx`, `Ny`, `Px`, and `Py` are all powers of two and may be defined as global constants. The initial conditions may be set any way you like: Either computed locally on each process or computed on one process and distributed.

Document your choices and strategy in `P2.pdf`.

HINT: Debug with an interactive session on a compute node with up to $(\texttt{Px}, \texttt{Py}) = (4, 2)$ before commenting out the `draw_now` lines (the `save_now` lines should still work) and running on the headnode with more processes.

HINT: In `Plotter3DCS205.py` we have provided two classes that will help you plot 3D data. The first class, `MeshPlotter3D`, is used in the provided serial version. If this class is used in an **interactive session on a compute node** then each process can launch and update a separate plot with its local data. Alternatively, if you use the `MeshPlotter3DParallel` class and pass in the global indices for your local data (i.e. On some process, `u[1,1]` might represent $u_{32,32}$. Here, $(1, 1)$ is the local index and $(32, 32)$ is the global index), then the data will be gathered and only a single plot will be produced with the global data. The script `Plotter3DCS205.py` also includes a simple example which implements `MeshPlotter3DParallel`, which can be viewed by running the plotting script itself in interactive mode:

```
$ mpirun -n (Px*Py) python Plotter3DCS205.py Px Py
```

Both `MeshPlotter3D` and `MeshPlotter3DParallel` can be used to efficiently find errors or impress a friend.

sendrecv

Make an implementation `P2A.py` that uses `Sendrecv` or `sendrecv`.

Isend/Irecv

Make an implementation `P2B.py` that uses `Isend/Isecv` or `isend/irecv`.

Analysis

With some chosen (`Nx`, `Ny`), use the headnode to measure the time/iteration and the speedup with various (`Px`, `Py`) pairs. Discuss the results.

Recall that weak scaling is exhibited when the problem size per processor is kept constant and the efficiency does not decrease. Does this problem exhibit weak scaling? Why or why not. Explain your reasoning with theoretical and/or experimental results.

HINT: If the amount of work per processor is kept constant, how does the amount of communication per processor scale as the number of processors is increased? How does the overhead scale?

Submission Requirements

- `P2A.py`: Completed `sendrecv` implementation.
- `P2B.py`: Completed `Isend/Irecv` implementation.
- `P2.pdf`: Explanations and/or plots.