HOMEWORK 3

CSPP51087 WINTER 2015
DUE:FRIDAY, FEB. 20

The advection equation is a hyperbolic partial-differential equation (PDE) governing the conserved movement of some pulse with some velocity. In 1D, it is written as

$$\frac{\partial C}{\partial t} + v\frac{\partial C}{\partial x} = 0 \tag{1}$$

where v is the velocity. In 2D, it is written as

$$\frac{\partial C}{\partial t} + u\frac{\partial C}{\partial x} + v\frac{\partial C}{\partial y} = 0 \tag{2}$$

where C = C(x,y,t) is some scalar concentration and $\vec{v} = (u, v)$ is the 2D velocity. In general $\vec{v} = \vec{v}(x, y, t)$ but in this homework we'll assume that the velocity is constant in space and time and can thus be represented by two scalar values.

Those who have taken a numerical analysis course are familiar with the techniques used for solving PDEs on computers. Those who haven't can find everything they need to complete the exercise in this document. In particular, explicit, finite difference approximations allow us to simulate the behaviors of the PDE on a discrete grid given some initial condition, $C(x, y, 0) \equiv C_0(x, y)$.

For example, the Lax method for numerically solving the 2-d advection equation can be written as:

$$C_{i,j}^{n+1} = \frac{1}{4}(C_{i+1,j}^n + C_{i-1,j}^n + C_{i,j+1}^n + C_{i,j-1}^n) - \frac{\Delta t}{2\Delta}[u(C_{i+1,j}^n - C_{i-1,j}^n) + v(C_{i,j+1}^n - C_{i,j-1}^n)] \tag{3}$$

where $C_{i,j}^n$ is the scalar field (e.g. temperature) at each physical location i, j on a discretized two-dimensional mesh, and n is the discrete time unit. Δt and Δ represent the time step and mesh spacing respectively, and (u, v) are the two components of the velocity field. Once the discrete initial condition $C_{i,j}^0$ is specified this equation can be iterated to get the new values $C_{i,j}$ at the next time step.

For the purpose of this exercise these details, however, are unimportant. Consider the right hand side of formula as the "kernel function", analogous to the smoothing operator in our class prototype. Recall that the kernel function is simply some code that computes updates to each point on a grid based on some combination of existing points. In this case, to compute the right hand side in parallel requires one-level deep of *ghost cells* (since each point depends on its four neighbors on a Cartesian mesh). Thus, to iterate the formulate above requires a two-dimensional ghost cell filling routine. As shown below it will also require global communication to test the time step condition.

In this exercise you will begin with a 2D Gaussian pulse initial condition and via the Lax scheme, integrate it forward in time it with some constant velocity. Be sure that the Courant stability condition

$$\Delta t \leq \frac{\Delta}{\sqrt{2}|\vec{v}|} \tag{4}$$

is satisfied.

1. Part 1

Write a serial code to solve the equation above. Here is the pseudocode for the algorithm. The domain has size $L \times L$ and is discritized on an $N \times N$ mesh. The simulation is iterated for a total time of T with N_T timesteps.

Algorithm 1 Serial 2D Lax-Wendroff solver

input N, N_T, L, T,
allocate $N \times N$ grid
set velocity field components u, v
$\Delta = \frac{L}{N}$
$\Delta t = \frac{T}{N_T}$
Assert $\Delta t \leq \frac{\Delta}{\sqrt{2}\sqrt{u^2+v^2}}$
Initialize $C^0_{i,j}$ for all i, j as a Gaussian
for $1 \leq n \leq N_T$ **do**
    for $1 \leq i \leq N$ **do**
        for $1 \leq j \leq N$ **do**
            if i, j is on boundary **then**
                Apply wrap-around boundary conditions
            end if
            update $C^{n+1}_{i,j}$ using The Lax method (equation 3)
        end for
    end for
end for

2. Part 2

Write a parallel code using a two-dimensional cartesian decomposition and blocking communication. The MPI cartesian communicator functions are helpful for setting grid, see `MPI_Cart_create()`, `MPI_Cart_shift()`, etc.

3. Part 3

Create a code in which the blocking send/recv calls are replaced by non-blocking communication.

4. Part 4

Do performance analysis/comparison between the blocking and nonblocking communication implementations.

5. Part 5

Create a shared memory version of the code using OpenMP

6. Part 6

Compare OpenMP to the best MPI timings for a range of parameter valies.
Include weak and strong scaling studies from the Midway platform.

Parallel code comments:

(1) Do not worry about making your code run for an arbitrary number of processors. It is ok to specify that it can only run on even numbers, powers of two, etc. But you most document it and do error checking within the program.
(2) Have your code accept an input parameter file (rather than argc/argv) to specify the simulation characteristics.