MPCS 51087
Problem Set 1

March 2018

1 Serial and Threaded Advection

Due Date: Tuesday, April 10, 2018 by 11:59 pm

The advection equation is a hyperbolic partial-differential equation (PDE) governing the conserved movement of some pulse with some velocity. In 2D, it is written as:

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

where $C(x, y, t)$ 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 many 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, $C(x, y, 0) = C_0(x, y)$.

In this homework, your task will be to numerically estimate the solution to Equation 1 using the Lax method. The goal is to take the initial 2D system state at $t = 0$ and to simulate how the system will develop over time. We can begin this by using the finite difference method, wherein the continuous spatial and temporal dependencies of Equation 1 are discretized. We can then iterate over discrete timesteps n, computing C^{n+1} based off of C^n using an explicit expression. To determine the expression used to move the system forward in time, we will first consider a forward time, center space (FTCS) finite difference approximation:

$$\frac{C^{n+1}_{i,j} - C^n_{i,j}}{\Delta t} + u\frac{C^n_{i+1,j} - C^n_{i-1,j}}{2\Delta x} + v\frac{C^n_{i,j+1} - C^n_{i,j-1}}{2\Delta y} = 0 \tag{2}$$

where $C^n_{i,j}$ is the scalar field (e.g. temperature) at each physical location i, j on a discretized two-dimensional mesh, n is the discrete time unit, Δx and Δy are the mesh spacings in the x and y directions respectively, and (u, v) are the two components of the velocity field. With the discrete initial condition $C^0_{i,j}$ specified, this equation can be iterated to get the new values $C^{n+1}_{i,j}$ at the next time step, by re-writing Equation 2 into the following form:

$$C^{n+1}_{i,j} = C^n_{i,j} - \frac{\Delta t}{2}\left[u\frac{C^n_{i+1,j} - C^n_{i-1,j}}{\Delta x} + v\frac{C^n_{i,j+1} - C^n_{i,j-1}}{\Delta y}\right] \tag{3}$$

We can further simplify things for this assignment by assuming that we have evenly discretized our spatial mesh in both dimensions, such that $\Delta x = \Delta y$, so that our FTCS finite difference expression becomes:

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

Finally, we will alter our FTCS equation further by way of the Lax method, wherein the term $C_{i,j}^n$ has been averaged by way of its four neighbors, as:

$$C_{i,j}^n = \frac{C_{i-1,j}^n + C_{i+1,j}^n + C_{i,j-1}^n + C_{i,j+1}^n}{4} \tag{5}$$

to reach our final Lax method explicit timestep expression of:

$$\boxed{C_{i,j}^{n+1} = \frac{1}{4}\left(C_{i-1,j}^n + C_{i+1,j}^n + C_{i,j-1}^n + C_{i,j+1}^n\right) - \frac{\Delta t}{2\Delta x}\left[u\left(C_{i+1,j}^n - C_{i-1,j}^n\right) + v\left(C_{i,j+1}^n - C_{i,j-1}^n\right)\right]} \tag{6}$$

One thing to note regarding the Lax method, is that the approximation made in Equation 5 adds in a dispersive effect into the advection problem. This is not great from a fidelity stand point, but for the purposes of the homework assignment the dispersive effect makes for more interesting plots.

1.1 Serial Implementation (20 pts)

Write serial code in a HPC language of your choice (C, C++, or Fortran) to implement the above scheme. Begin with a 2D Gaussian pulse initial condition:

$$C_{i,j} = \exp\left(-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)\right) \tag{7}$$

where (x_0, y_0) is the center and σ_x and σ_y is the "spread" of the Gaussian blob in each dimension. Once initialized, use the Lax scheme to integrate the 2D domain forward in time with some constant non-zero velocity. Use periodic, or "wrap-around", boundary conditions. Also, be sure that your code asserts that the Courant stability condition:

$$\Delta t \leq \frac{\Delta x}{\sqrt{2}|\vec{v}|} \tag{8}$$

is satisfied, so as to avoid numerical instability.

The pseudo code for the algorithm you will be implementing is given in Algorithm 1.

Algorithm 1 Advection Pseudocode

```
Input N, NT, L, T, u, v
Allocate N × N grid for C^n_{i,j}
Allocate N × N grid for C^{n+1}_{i,j}
Δx = L/N
Δt = T/NT
Assert Δt ≤ Δx/√(2(u²+v²))        ▷ Ensure parameters meet the Courant stability condition
Initialize C^n_{i,j} for all i, j as a Gaussian        ▷ Equation 7
for 1 ≤ n ≤ NT do
    for 1 ≤ i ≤ N do
        for 1 ≤ j ≤ N do
            if i, j is on boundary then
                Apply periodic boundary conditions
            end if
            Update C^{n+1}_{i,j}        ▷ Using Equation 6
        end for
    end for
    Set C^n_{i,j} = C^{n+1}_{i,j} for all i, j        ▷ Or, swap references
end for
```

You should read in your inputs for N, NT, L, T, u, and v as command line arguments, in that order. This will allow us to experiment and test your code when grading or if we are helping to debug. You can hard code the Gaussian parameters, however. When you run your code, it should write to stdout the parameters it will simulate before beginning, as well as an estimate for the amount of memory required (when using double precision) to perform the simulation. Some good input parameters for testing and development that meet the Courant stability condition are:

- N = 400 (Matrix Dimension)
- NT = 20000 (Number of timesteps)
- L = 1.0 (Physical Cartesian Domain Length)
- T = 1.0e6 (Total Physical Timespan)
- u = 5.0e-7 (X velocity Scalar)
- v = 2.85e-7 (Y velocity Scalar)

which would be input to your program when running as:

```
./my_advection_program 400 20000 1.0 1.0e6 5.0e-7 2.85e-7
```

1.2 Plotting (10 pts)

Your will need to be able to visualize your data in order to ensure that your code is operating as expected. In HPC, it is common to output a file to disk containing data of interest that can be manipulated, analyzed, and plotted later. For this homework assignment, implement a serial function to write your 2D domain data for a single timestep to file in ASCII format. Use scientific notation out to a couple of decimal places, and store the data in 2D matrix format. Do not worry about storing any Cartesian x,y location information to file, just store the contents of the matrix.

Then, using the graphing program or library of your choice, write a script that plots your data file. Some suggestions might be python, gnuplot, matlab, or julia.

To give you a point of reference for what you are looking for, we have made an animated example:
`http://i.imgur.com/J1b0JUG.gif`
which shows the expected results when using the default parameters given above, along with an initial Gaussian centered at the middle of the problem domain with $\sigma_x = \sigma_y = \frac{L}{4}$. Frames are taken every 1000 timesteps.

For this assignment, you will need to generate at least three plots. One showing your initialized Gaussian distribution, and another two plots at later timesteps that clearly show the development of the advection/dispersive process over time. Include all three of your plots in your PDF write-up for the assignment.

1.3 Shared Memory Threading (20 pts)

Parallelize your advection code using a shared memory threading model, via OpenMP. It is highly recommended that you complete the serial implementation and plotting sections first, and thoroughly debug your program before proceeding. Also, you will need to submit separate versions of the serial and parallel codes when you turn them in. This should be done by creating a copy of your serial code into a separate folder, and then parallelizing the new version. More details on how exactly we would like your code structured given later on in the "What to Submit" Section.

A few notes on your implementation:

- Use the `omp_get_wtime()` function to measure the amount of time your code takes to run.
- You will be performing scaling studies using a variety of thread counts. Telling your code the number of threads to use can be done in a variety of ways. For this assignment, we would like you to do this by taking in as an additional command line input the number of threads to run. Using this command line argument, you can use the function `omp_set_num_threads()` in your code to specify how many threads to use.

- You are required to use the `default(none)` tag when entering the parallel region of your code, which ensures that you will be forced to declare all variables as either private or shared.
- Be sure to parallelize your Gaussian initialization routine as well.
- Do not parallelize by time steps. In a computation like this, you cannot begin calculating timestep n+1 without knowing what the grid looks like at n first. I.e., You should parallelize work being done within time steps, not the time steps themselves.
- Do not make this assignment more complicated than is necessary – you should be able to accomplish parallelization of this algorithm via OpenMP by adding only a few lines of code.

1.4 Performance Analysis

1.4.1 OpenMP Thread Scheduler Performance Testing (10 pts)

Experiment with the various thread scheduling techniques available in OpenMP, while running on at least 16 physical CPUs, using Midway resources if needed. In particular, try the `dynamic`, `static`, and `guided` options. Experiment with work block sizes. Include in your write-up PDF a few sentences regarding your findings and observations, specifically:

- Overall, do you notice any difference between the various options & block sizes, and if so, which configuration is optimal?
- What other use cases in numerical simulation or HPC might see a different (greater or lesser) performance impact from these OpenMP options?

There is no need to do a rigorous parameter sweep of all possible schedulers, work block sizes, and problem sizes, just some tinkering is fine.

1.4.2 Strong Scaling Study (15 pts)

Strong scaling studies are a common way to measure the parallel efficiency of a code when running a fixed *total* problem size. Strong scaling plots typically display the observed parallel "speedup" as compared to the ideal. A strong scaling study can be generated by running the code with the same configuration while varying the number of threads the code executes with. Record the runtime for the code for each run. Then, compute the parallel speedup, where speedup is defined as:

$$\text{Speedup} = \frac{\text{Recorded Runtime with 1 Thread}}{\text{Recorded Runtime With } N \text{ Threads}} \tag{9}$$

The ideal speedup for N threads is therefore simply N. For instance, if a code were run in serial and takes 10 seconds, and when run on two cores takes 5.1 seconds, we find that it has a parallel speedup of 1.96 (compared to the ideal speedup of 2).

Do a strong scaling study of your code out to at least 16 physical CPUs, and plot the observed vs. ideal speedup (with threads on x-axis, and speedup on y-axis). Use the following input variables for your strong scaling study:

- N = 3200
- NT = 400
- L = 1.0
- T = 1.0e3
- u = 5.0e-7
- v = 2.85e-7

Then, perform a second strong scaling study, but this time with a much smaller N = 200.

Along with your two strong scaling study plots, include in your write up PDF a few sentences on what the plots show regarding your code and the parallelizability of the two problem sizes you studied. If your results are less than (or better than) ideal, offer some ideas as to what might cause this.

1.4.3 Weak Scaling Study (15 pts)

Weak scaling studies are a common way to measure the parallel efficiency of a code when running a fixed problem size *per core*. I.e., as you test with more threads, you also make the total problem size larger, but maintain the same problem size per thread ratio.

A weak scaling study can be generated by running the code with a problem size configuration while varying the number of threads the code executes with. For instance, if running a 100×100 matrix with 1 thread, you will need to run a 141×141 matrix for 2 threads. Record the runtime for the code for each run.

Do a weak scaling study of your code out to at least 16 physical CPUs, and plot the observed vs. ideal runtimes (with threads on x-axis, and runtime on y-axis). In the ideal case, the code will take the same overall runtime to complete regardless of how many threads are used, so the ideal weak scaling plot is flat, with an ideal runtime always equivalent to the observed serial case.

Use the following input variables for your weak scaling study:

- N = 800 (increase as you add threads)
- NT = 400
- L = 1.0
- T = 1.0e3
- u = 5.0e-7
- v = 2.85e-7

Along with your weak scaling plot, include in your write up PDF a few sentences on what the plot shows regarding your code and if your code is capable of efficiently utilizing your test computer's resources. If your results are less than (or better than) ideal, offer some ideas as to what might cause this.

1.5 Compiling, Code Cleanliness, and Documentation (10 pts)

We plan on compiling and running your code as part of grading it. You must include a makefiles (one for your serial and one for your parallel version) capable of compiling your code into a runnable executable. Also, while we are not grading to any rigid coding standard, your code should be human readable and well commented so we can understand what is going on. As you will be building on this code for the next assignment, it will be useful to have a clean code base to build off of.

1.6 What to Submit

You submission should be in the form of a zip or tar file containing the following items:

- Your source code. You need to submit separate serial and threaded versions of the source code. You should have two folders in your submittal, each one containing all the source code for the two different versions.
- Working makefiles, one in each directory (serial & parallel), so that we can easily compile and run your code
- Your plotting script(s)
- A single writeup (in PDF form) containing all plots and discussions asked for in the assignment