Topic #6: Realizing a stencil

Overview:

Objective: To implement a simulation of heat diffusion on a 2D square grid using an MPI stencil

Roadmap:

This module consists of 3 activities:

#1: Implement a sequential simulation of the problem
#2: Implement a parallel MPI simulation using a stencil with 1D topology
#3: Implement a parallel MPI simulation using a stencil with 2D topology

Activity #1:

In this activity we implement a sequential simulation of heat diffusion.

We consider a 2D square system (x and y coordinates, each ranging from 0 to an arbitrary L value) with an initial heat repartition (that is, a given value for the temperature T(x,y) for every (x,y) point at the beginning of the simulation). The heat will diffuse over time, according to the following equation:

$$\frac{\partial T}{\partial t}(x,y,t) = D\left(\frac{\partial^2 T}{\partial x^2}(x,y,t) + \frac{\partial^2 T}{\partial y^2}(x,y,t)\right)$$

where D is a diffusion coefficient, set to a constant value (you may choose any value, as we will abstract computations)

The resolution of such a problem requires to choose a boundary condition. In our case, this simply means that T will remain constant on the boundaries of the domain:

For every $x,y \in [0,L]$, for every t:

$$T(x,L,t) = T(x,0,t) = T(L,y,t) = T(0,y,t) = T_0$$

where T_0 is the temperature on borders (can also be set to any value here)

Let us now discretize our square domain with a squared mesh. Considering an arbitrary number s, we divide our square domain into s^2 squares, each one being L/s by L/s, and each one having a single temperature value. Let us now note the coordinates of a square as (i,j), where i and j are integers ranging from 0 to s-1. We now refer to the temperature as $T(i,j,t)$.

It is possible to compute $T(i,j,t+1)$ from knowing $T(i,j,t)$ and its four neighbors at instant t, namely: $T(i-1,j,t)$, $T(i,j-1,t)$, $T(i+1,j,t)$ and $T(i,j+1,t)$. You are provided with a helper function:

double update_temperature(double current_case, double i_m1, double j_m1, double i_p1, double j_p1, double D, double T0, double dx, double dt)

that takes as first 5 parameters the current case value and its 4 neighbors values (in the order described above), as 6th and 7th parameters D and T0, and as 2 last parameters, the space step and the time step, and returns the value of $T(i,j,t+1)$. The space step is the space between 2 squares, that is, L/s. The time step is the time between two consecutive grids of temperature repartition. Assuming we want to simulate diffusion over a period of time τ, where the temperature over the grid is updated r times, the time step is then τ/r

This function computes temperature according to the differential equation shown above. (It computes the heat diffusion equation shown above).
Note that this function cannot compute the temperature for any point that is on the border of the domain, as it has less than four neighbors. That isn't a problem, because as we said, the temperature is set to a constant and need not to be computed for the border squares.

What to do:

Step #1:

-Implement a C (or C++) program called sequential_heat_diffusion(source sequential_heat_diffusion.c) that:

- Takes a single command-line argument, s, an integer that's strictly positive.
- Allocates a 2-D double array that is s by s
- Fill this array with temperature values corresponding to an initial temperature repartition (don't forget to create a T0 variable and set the temperature on the borders to T0)

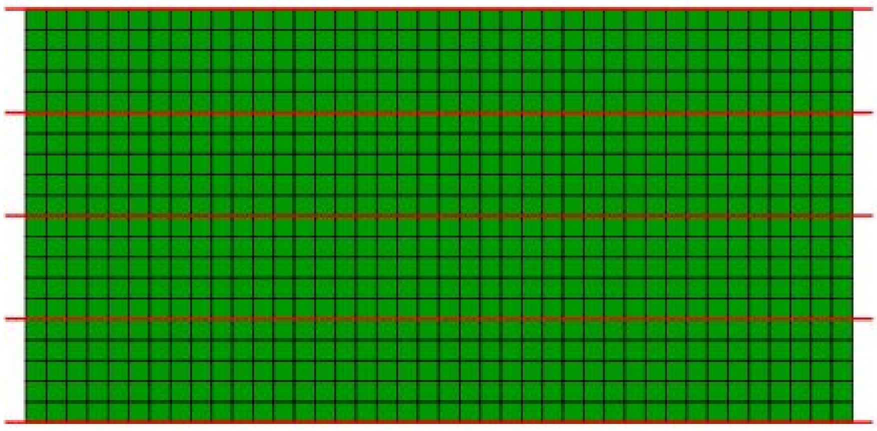
Step #2:

-Add the simulation of diffusion to your program.

- Set a period of time tau, a number of instants r, a diffusion coefficient D, a lenght L as variables in your main.c
- Realize the simulation of the diffusion by updating N times the temperature grid with the helper function provided

Activity #2:

Implement a parallel version of your program from activity #1, using a 1D MPI stencil. This means that each process has to update a certain chunk of contiguous rows (or columns, depending on what suits you best and how you defined your n by n array), the size of each chunk must be evenly shared among processes. For simplicity, we will assume that the number of process n divides s, which makes it possible to have the exact same number of rows for each process. Below is an example of such a data distribution:



The red lines delimitate the chunks for each process

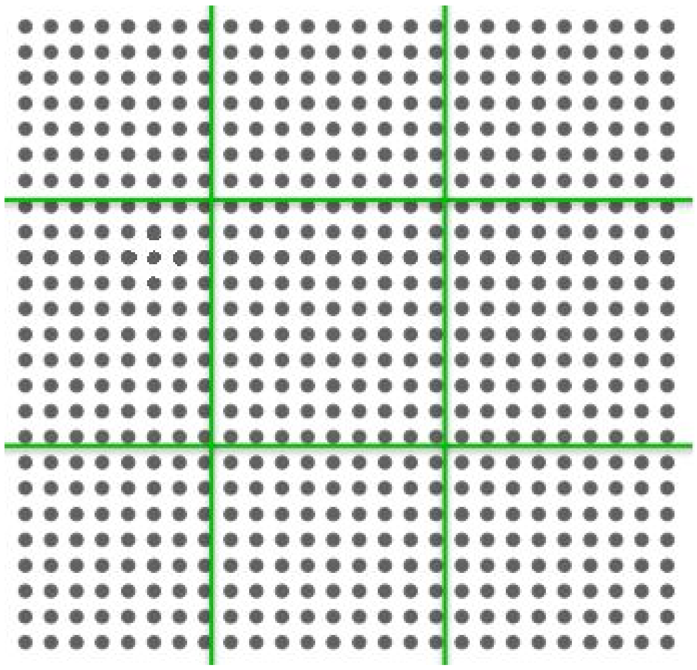
What to do:

Implement a parallel MPI program that takes 2 command lines arguments n and s. Each process from 1 to s should set the initial temperature values for its squares. Then, for each time step, all processes update their values. Note that, as the computation of a square requires to know the values of the neighboring squares, each process will have to exchange some of its values with its neighbor(s) process(es) in order to be able to update his own values. Consequently, all processes will have to wait for each other at each time step.

Activity #3:

#Step 1

Implement a parallel version of your program using a 2D MPI stencil. This time, each process handles a square part of the L by L domain. For simplicity, you may assume that the number of processes is a perfect square s^2, where its square root s divides n. Thus, squares are evenly distributed among processes. (Each process will have a “big square” of several small squares). An illustration of this data distribution is shown below.


The green lines delimitate the chunks allowed to each process.

This time, each process has to communicates with the four (or less) others neighboring processes so that all processes can update their values.
For this part of the activity, we will use synchronous communications only. In the following part, we will use asynchronous communications.

#Step 2

Modify your previous program to now use asynchronous communications. Compare the execution time between the two implementations for various values of the latency and transfer rate. What is the limit transfer rate above which synchronous communications become faster ?

Sources:

- https://en.wikipedia.org/wiki/Heat_equation
- https://en.wikipedia.org/wiki/Von_Neumann_stability_analysis
- Stencil lesson (cf “Sources” folder)