Solving 2D Laplace equation using Jacobi Iteration method

1 Description of the problem

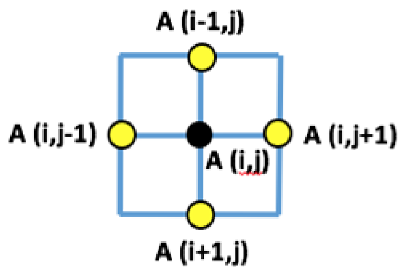
The **2D Laplace equation** is important in many fields of science, notably the fields of electromagnetism, astronomy, and fluid dynamics (as in the case of heat propagation).

$$\nabla^2\Phi(x,y) = \frac{\partial^2\Phi(x,y)}{\partial x} + \frac{\partial^2\Phi(x,y)}{\partial y} = 0$$

The partial differential equation, or PDE, can be discretized, and this formulation can be used to approximate the solution using several types of numerical methods.

$$\frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{h^2} + \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{h^2} \approx 0$$

$$\Phi_{i,j} \approx \frac{1}{4}[\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}]$$



The **iterative Jacobi** scheme or solver is a way to solve the 2D-Laplace equation. Iterative methods are a common technique to approximate the solution of elliptic PDEs, like the 2D-Laplace equation, within some allowable tolerance. In the case of our example, we will perform a simple **stencil calculation** where each point calculates its value as the mean of its neighbors' values.

The stencil is composed of the central point and the four direct neighbors. The calculation iterates until either the maximum change in value between two iterations drops below some tolerance level or a maximum number of iterations is reached.

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

2 Description of the provided code

We provide you with a basic implementation of the Laplace method. We will assume as an input 2D matrix, A, with fixed dimensions **n** and **m**, defined as a global variable (outside any function). Initial data determine the initial state of the system. We assume that all the interior points in the 2D matrix are zero, while the boundary state, which is **fixed** along the Jacobi Iteration process, is defined as follows:

```
// boundary conditions for up and down borders
For each j from 0 to m:
        $A_{0,j} = A_{n-1,j} = 0;$

// boundary conditions for left and right borders
For each i from 0 to n:
        $A_{i,0} = sin(i * \pi/(n-1)$  $A_{i,m-1} = sin(i * \pi/(n-1) * e^{-\pi};$
```

The outermost loop that controls the iteration process is referred to as the **convergence loop**, since it loops until the answer has converged by reaching some maximum error tolerance or number of iterations. Notice that whether or not a loop iteration occurs depends on the error value of the previous iteration. Also, the values for each element of A are calculated based on the values of the previous iteration, known as a **data dependency**.

The first loop nest within the convergence loop calculates the new value for each element based on the current values of its neighbors. Notice that it is necessary to store this new value into a different array, or auxiliary array, that we call **Anew**. If each iteration stores the new value back into itself then a *data dependency exists* between the data elements, as the order of each element is calculated affects the final answer. By storing into a temporary or auxiliary array (Anew) we ensure that all values are **calculated using the current state of A before A is updated**. As a result, *each loop iteration is completely independent of each other iteration.* These loop iterations may safely be run in any order and the final result would be the same.

A second loop calculates a maximum error value among the errors of each of the points in the 2D matrix. The error value of each point in the 2D matrix is defined as the square root of the difference between the new value (in Anew) and the old one (in A). If the maximum amount of change between two iterations is within some tolerance, the problem is considered converged, and the outer loop will exit.

The third loop nest simply updates the value of A with the values calculated into Anew. If this is the last iteration of the convergence loop, A will be the final, converged value. If the problem has not yet converged, then A will serve as the input for the next iteration.

Finally, every ten iterations of the converge loop the actual error is printed on the screen to check that the execution is progressing correctly, and the error is converging.

3 Recommendations for an MPI solution

As you know, sometimes a real problem may be too big (in size or complexity) to be tackled in a single node. In this case, we can develop a parallel distributed solution that makes use of multiple nodes for solving the problem in a reasonable time.

In this assignment, we are going to focus on parallelizing Laplace problem using MPI as the communication and synchronization mechanism. The application will be able to use distributed memory and be executed on different nodes exploiting, in this way, the potential parallelism of the work distribution.

You have the Laplace program available in a sequential version. Suppose now that we are going to use N processes to solve the problem in a distributed way. We can assume that the number of rows is divisible by number of processes. According to the characteristics of the problem, each of these processes will have to carry on the computation over a portion of the data (part of the matrix), taking care of interchanging the necessary data and synchronizing with other processes. The specific processes which will have to communicate will depend on how the data is partitioned among them.

The use of memory for each process can be reduced proportionally to the number of processes with the addition of some extra rows in each process for storing the data (border rows) received from its neighbors. If the sequential algorithm takes a time T to complete, we can expect that the time of this distributed version could be T/N + Communication-overhead. It is worth noticing that the extra amount of memory used, and the communication overhead would increase their impact on the applications' performance as the number of processes used increases.

The assessment of a parallel application performance usually consists in analyzing the scalability of the application (changing the amount of data to be processed or the number of resources used) and explaining the causes of the observed results. This means that you should execute the application a) using different input sizes (different matrix size), b) varying the number of resources, and hence calculate the speedup and efficiency (**strong scalability**) and c) do the same varying both parameters at the same time - the input size and number of resources (**weak scalability**). You can also try to quantify the communication overhead, the communication/computation ratio, or the functions that are consuming more time, to explain the application behavior.

Make a performance analysis of your program. **NOTE:** using perf on a

MPI application is meaningless (as it gives us the data only for one process), use MPI_Wtime instead.

4 Environment for testing and experimental work

Remember that you must do the executions using the SLURM job manager. You can prepare 2 SLURM submission scripts:

- For testing and checking results – compilation and sequential execution for a small input data (this version gives you the correct results); compilation and parallel execution for a small input data to check if the parallel version gives correct results.
- For final experimentation and time collection – compilation and sequential execution (the execution time of this version serves as a reference time); compilation and exhaustive parallel execution: different number of processes and different input data (different matrix sizes) for analyzing the performance.

IMPORTANT: you can use both clusters, aolins and Wilma for testing and experimental work.