Computer Architecture and Operating Systems Department (CAOS/DACSO)

Engineering School

Universitat Autònoma de Barcelona

Course: Parallel Programming

C Programming

Deliverable

Useful commands to get started

Connect to cluster:

ssh -p 54022 ppm-60@aolin-login.uab.es

Compilation:

gcc -Wall -g myProgram.c -o myProgram

Run:

./myProgram

For more details, refer to the documentation of the campus virtual.

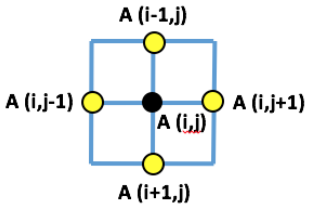
Laplace equation

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

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

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.

$$\left(\frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{h^2}\right) + \left(\frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{h^2}\right) \approx 0 \quad \Phi_{i,j} \approx \frac{1}{4}\left[\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}\right]$$



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.

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

We will assume 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:

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

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

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 should calculate 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 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 must calculate 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 must simply update 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 must be printed on the screen to check that the execution is progressing correctly, and the error is converging.

1. Write a C code to solve the previous problem. Assume that n and m are constant values. Use single precision floating-point values to represent the values of the simulated system.
2. Modify the code so that parameters n and m are provided at runtime in the command line, and the memory space for the data used in the simulation is dynamically allocated.
3. Modify the code so that it is functionally equivalent, but it is executed faster. This is called *program optimization.*
4. Execute the program with different number of n and m (for example, 100, 1000, 4096) and measure the execution time (you can use perf program).

Program Structure

```
// define n and m
// declare global variables: A and Anew

int main(int argc, char** argv)
{
  // declare local variables: error, iter_max ...
  // get iter_max from command line at execution time
  // set all values in matrix as zero
  // set boundary conditions

  // Main loop: iterate until error <= tol a maximum of iter_max iterations
  while ( error > tol && iter < iter_max ) {
    // Compute new values using main matrix and writing into auxiliary matrix
    // Compute error = maximum of the square root of the absolute differences
    // Copy from auxiliary matrix to main matrix
    // if number of iterations is multiple of 10 then print error on the screen
  } // while
}
```