

Computer Architecture and Operating Systems Department (CAOS/DACSO)

Engineering School
Universitat Autònoma de Barcelona

LAB: Parallel Programming

MPI Programming Assignment

Lab Exercises



Introduction

During this course, you have been working on programming, optimizing and parallelizing the solution to different problems. Up to now, optimization and parallelization have been focused on efficiently taking advantage of the resources of a single compute node. However, sometimes a 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.

We are going to have two lab sessions focused on parallelizing the solution to the 2D Laplace equation on a cluster of nodes, using MPI as the communication and synchronization mechanism.

The main idea is to propose and implement a basic distributed version of this program and then to analyze its performance and discuss (maybe also implement) possible improvements.

First, we refresh the discussion about the iterative method for solving the Laplace equation, with the objective of having all the information in the same document. You have probably already implemented and optimized this program, but, in any case, this exercise assumes that we start from the code version that can be downloaded along with this document. Next, we present an idea for partitioning the problem in the simplest way and an associated simple algorithm structure. Implementing this solution is the minimum necessary for passing this assignment. Then, we list several possible optimizations for this initial solution and ask you to implement one of them. Implementing one optimization worth up to 3 points of the assignment. Finally, we hint some criteria to assess your program versions. Getting performance information with the tool used in the lab (extra lab session) and presenting it along with a reasoned interpretation will make up for the remaining 2 points for this assignment.

Program: Solving 2D Laplace equation using Jacobi Iteration method

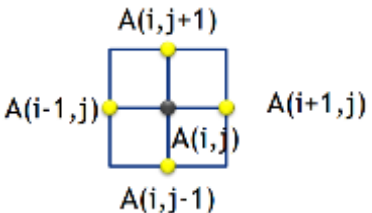
The **2D Laplace equation** is important in many fields of science, notably the fields of electromagnetism, astronomy and fluid dynamics (like in the case of heat propagation).

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

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.

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

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.

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


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 : $A_{i,0} = \sin(i * \pi / (n-1))$ $A_{i,n-1} = \sin(i * \pi / (n-1)) * e^{-\pi}$ // 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.

Basic Parallel Solution

Suppose that we are going to use N processes to solve the problem in a distributed way. Accordingly with the characteristics of the problem, each of these processes will have to carry on the computation over a portion of the matrix A , taking care for interchanging the necessary data and synchronizing with other processes. The specific processes that will have to communicate will depend on how the data is partitioned among them.

The simplest partition of A is shown in Figure 1. In this case, each process P_i takes care of the computation of m/N rows of A and, in the general case ($0 < i < N-1$), it needs the last row of P_{i-1} and the first of P_{i+1} .

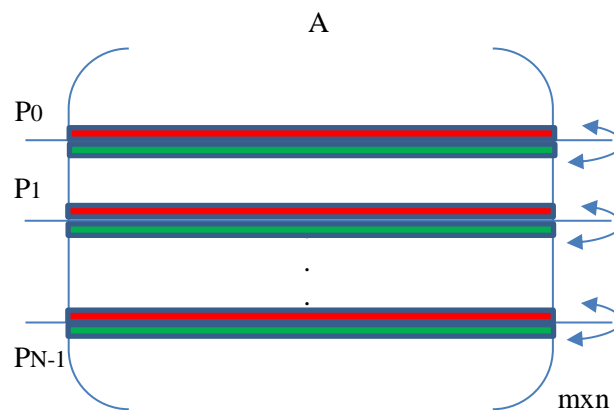


Figure 1

Using this scheme, and assuming that m is divisible by N (you can assume this in your solution), the basic solution algorithm could be like this:

1. MPI_Init()
2. me = Comm_Rank()
3. nprocs = Comm_Size()
4. Determine initial (ri) and final (rf) rows indexes
5. Allocate memory for my_A and my_temp (size $m/nprocs + 2$)
6. Init my_A and my_temp
7. while error > tol*tol && iter < iter_max do
 - if me > 0 then

```

    send( A[ri], me - 1)
    recv( A[ri-1], me - 1 )
    if me < nprocs-1 then
        send( A[rf], me + 1)
        recv( A[rf+1], me + 1)
    for ( j=ri; j < rf; j++ )
        for ( i=1; i < n-1; i++ )
            out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
            my_error = max_error( error, out[j*n+i], in[j*n+i] );
    Interchange and determine global error

```

As it can be seen, it is almost the same algorithm implemented in the provided code, just distributing the work and adding the data interchange among processes.

The use of memory for each process is reduced proportionally to the number of processes with the addition of two extra rows in each process for storing the data received from its neighbours.

Assuming that the sequential algorithm takes a time T to complete, we can expect that the time of this distributed version could be $T/N + \text{Communication-overhead}$.

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.

Task 1: Implement a working version of this algorithm (remember that you can assume that m is divisible by N). The code must be properly documented. You must include all the necessary information for running your code and indicate the test cases you have used.

Improving the Solution

The main limiting factor to the performance of the proposed solution is the overhead introduced by communication. Consequently, we should look for mechanisms that reduce this overhead.

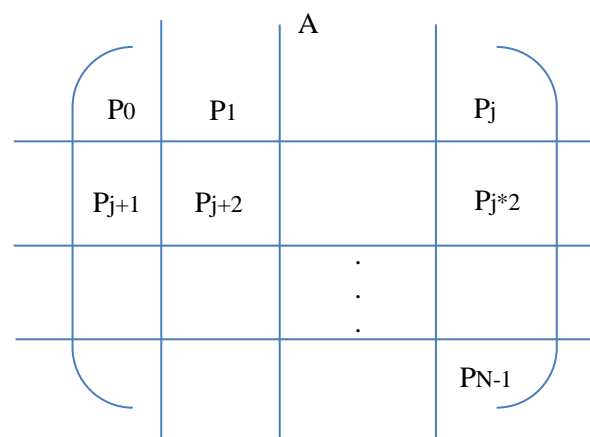


Figure 2

1. **Overlapping communication and computation.** We can change our algorithm for launching asynchronous communication calls and compute first the inner points of the matrix portion assigned to each process. After that, we can compute the border points, which depend on the data received from other processes.
2. **Block partitioning.** Instead of distribute A by rows, we can do a block distribution as the one shown in Figure 2. In this way the amount of communication between processes is reduced when the number of processes increases.

3. **Block communications.** Instead of interchanging the first and last rows for each process in each iteration, we can interchange the first and last k rows ($k \ll m/N$) and do k iterations (at the cost of increasing the number of elements computed in each process by $2 \cdot (k-1) \cdot n$).
4. **Hybrid solution.** Start only one process in each node and use OpenMP to parallelize the process over the available cores. In this case, you should take also good care of the submitting script for correctly managing the available resources. You must assure that each process has several cores exclusively available for launching the OpenMP threads.

Implementing any of these optimizations will make the solution code more complex. Of course, all these mechanisms can be combined, but consequently, the implementation difficulty significantly increases with each added optimization.

Task 2: Implement a working version of one of these mechanisms (you can make assumptions about the matrix dimensions $[m, n]$ and the number of processes N if needed). You must justify your selection, which means that you must discuss the perceived pros and cons of all methods. The code must be properly documented. You must include all the necessary information for running your code and indicate the test cases you have used.

Assessing your Solutions

It is likely that for this task you have two versions of your parallel program, the basic and the optimized one. However, if you only have the basic one, you can also assess its performance and earn this task's points.

The assessment of a parallel application performance usually consists in analyzing the scalability (strong and weak) of the application and explaining the causes of the observed results.

This means that you should execute the application using different 'fixed' input sizes and varying the number of resources and calculate the speedup and efficiency, and do the same varying both the input size and number of resources.

Then, depending on these results, you can try to quantify the communication overhead, or the communication/computation ratio, or the functions that are consuming more time, in order to explain them.

If you have implemented the two versions of the application, you should compare both of them, which is the speedup obtained with the optimized version over the basic one? Which is the tendency of the speedup when increasing the size of the problem/number of resources? Why?

Task 3: Make a performance analysis of your program versions using the given hints and the support of the performance analysis tools available in the lab. You must present an organized explanation of this analysis.

NOTE: using *perf* on a MPI application is meaningless, use TAU and MPI_Wtime instead.