Implementing the Numerical Solution to the Laplace Equation on a distributed-shared memory architecture using MPI

Gaurav Saxena (gaurav.saxena@bsc.es)

February 19, 2025

Keywords: Laplace, Finite Difference Method, MPI (Message-Passing Interface), HPC, Distributed-shared memory programming, Cartesian Topology, Stencil, overlapping, non-blocking, halos/ghost regions, C/C++/Python/Fortran, Dirichlet, Boundary-Value problem, Red-Black Gauss-Seidel, Vector, Contiguous.

1 Introduction

The *Laplace* equation is a second order, elliptic Partial Differential Equation (PDE) which is a special case of the *Poisson's* equation. For a scalar field ϕ, it can be written as:

$$\nabla^2 \phi = 0$$

. For two dimensions i.e. $\phi = \phi(x, y)$, on a square domain $\Omega = [0, 1] \times [0, 1]$, with equal mesh spacing $h = \frac{1}{N}$ in both dimensions, the Laplace equation, using a second order central difference scheme can be discretized using the *Finite Difference Method* (FDM) as:

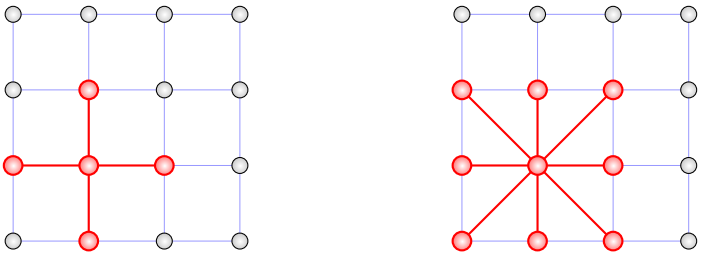
$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{h^2} = 0$$

where $\phi_{i,j} \equiv \phi(x_i, y_j)$ and $x_i = ih$, $y_i = jh$ for $i, j = 0, 1, 2, ..., N$. Rearranging the equation above, we get:

$$\phi_{i,j} = \frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}}{4}$$

.

Further, given the Dirichlet boundary condition $\phi = g(x, y)$ on the boundary Γ, we can obtain the numerical approximation of $\phi(x, y)$ on the domain by starting with an initial guess and iterating through the solution given by the equation above. In 2-D, this translates to taking the average of the values of the 4 neighboring points of the point (i, j). This pattern is known as a 5-pt



(a) 5-pt stencil (b) 9-pt stencil

Figure 1: Common stencils in 2-D

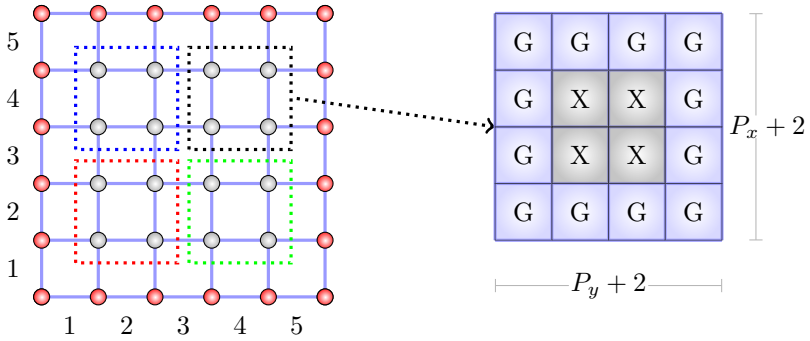


Figure 2: A vertex-centered problem of size $N_x \times N_y = 5 \times 5$, having 4×4 internal mesh points (unknowns, in grey color) is partitioned among 4 cores (shown by red, green, blue and black dotted lines). The result is a $(P_x + 2) \times (P_y + 2) = (2 + 2) \times (2 + 2)$ sub-domain (shown on right) with 4 original unknowns 'X' in that sub-domain and added ghost layer cells 'G'.

stencil as it involves the central point and 4 of its neighbors. Figure 1 shows the common stencils in 2-D.

2 Implementation

1. The domain $\Omega = [0, 1] \times [0, 1]$ is divided into $N_x \times N_y$ (Here, $N_x = N_y = N$) blocks of equal length and height $h = \frac{1}{N}$.

2. Figure 2 shows a domain divided into 5×5. The unknowns are indicated as grey circles and the boundary values as red circles in Figure 2.

3. A 2-D Cartesian topology of MPI processes equally divides the unknowns (MPI Topology is 2×2 in the figure).

4. Assuming the MPI topology is $D_x \times D_y$, each MPI process obtains $P_x \times P_y$ sized sub-domains where $P_x = \frac{N-1}{D_x}$ and $P_y = \frac{N-1}{D_y}$ (as the axes of the

coordinate system followed by MPI is different from the normal Cartesian Coordinate axes directions, we will adjust D_x and D_y when we implement the Cartesian topology[1]).

5. Notice that the numerical solution at point (i, j) involves its 4 neighbors in the NEWS direction and thus forms a 5-pt stencil.

6. For points (unknowns) at the boundary of a sub-domain (but not the actual physical boundary), we need values of the current solution from neighboring processes and hence we will use the MPI non-blocking communication calls `MPI_Isend()` and `MPI_Irecv()`. The advantage of using the non-blocking calls is that the computation and communication can be overlapped with each other.

7. The program takes into account the successive difference between the solution values at a point and continues till this becomes smaller than a tolerance. Thus, it might not converge but the idea is to observe the changes.

8. The language of implementation can be C/C++/Python/Fortran.

Figure 3 (read from top-left to bottom-right) shows how the solution evolves (not necessarily till convergence) using a single process and 4 processes. It can be seen that Figures 3a and 3b are identical.

3 Caution

We need to be extremely careful about the (a) direction of the X and Y axes of the domain and (b) direction of X and Y axes of the MPI Process Topology as these two will determine the (x, y) coordinates of a certain point. A convenient convention is to use the following (you can use any convention as long as you are consistent and do logical things):

1. The origin of the domain is at the top-left corner.

2. The X-axis of the Cartesian Coordinate system goes from left to right.

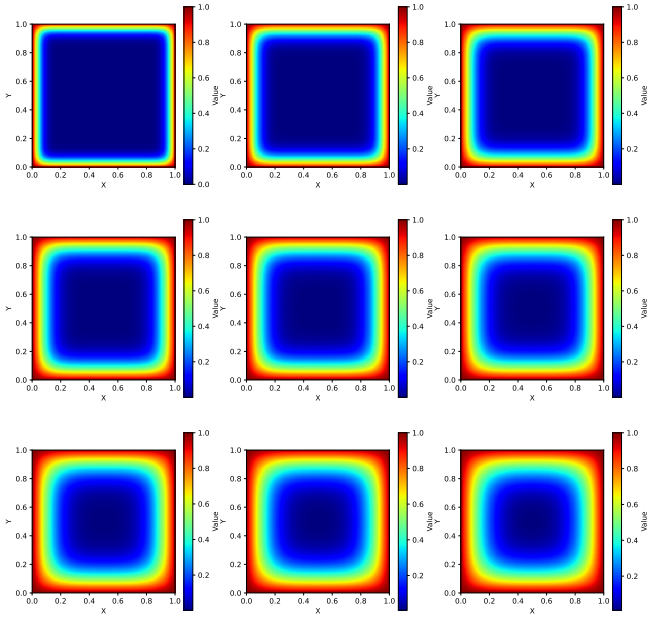
3. The Y-axis of the Cartesian Coordinate system goes from top to bottom.

4. The origin of the MPI Cartesian Topology is at top-left.

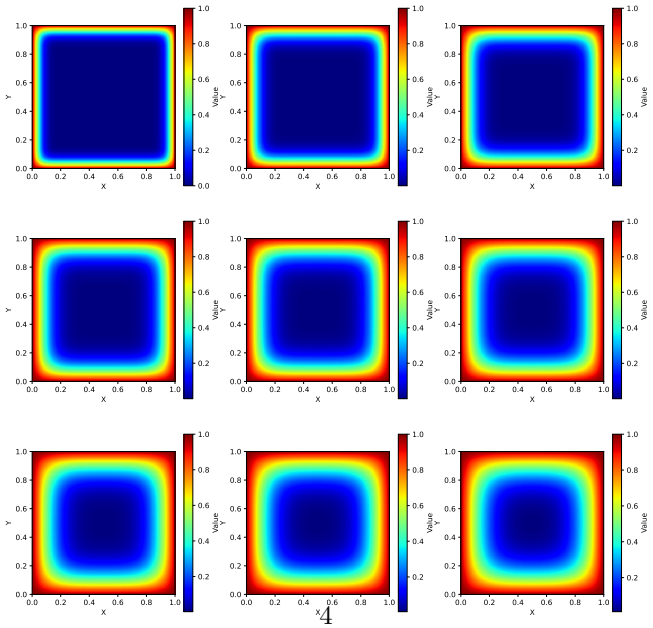
5. The X-axis of the MPI Topology goes from top to bottom.

6. The Y-axis of the MPI Topology goes from left to right.

[1] A Cartesian Topology is a virtual geometric arrangement of MPI processes that has nothing to do with their physical placement on the cores or nodes



(a) Solution using 1×1 processes



(b) Solution using 2×2 processes

Figure 3: Comparison of solutions obtained using 1 (1x1) and 4 (2x2) processes

4 Aim and Enhancement

The first aim is to understand the problem, its theoretical solution and the implementation using MPI. After you complete, we can suggest the following enhancements/directions (depending on your interest and time):

1. Visualize the solution in the domain after 1000, 2000, 3000, (choose appropriate iterations i.e. specify the value of `SAVE_ITER` in the C source file)... iterations using a heat-map (or any other graphical method). For this you will need the x, y coordinate of the point and the current value of the solution at that point (a python source file named `multiple_plots.py` is provided which should be able to plot all the `.dat` files in the current folder and save it as `.png` and `.pdf`).

2. Try to vary the boundary conditions[2] to observe change in solution. For e.g., the value of the scalar variable $\phi(x, y) = 1+\sin \pi x$ at the top boundary (or any other condition that you want to specify).

3. Use OpenMP for intra-node parallelization (so that we can have less processes, hence less communication) and see if the MPI + OpenMP version can be made faster than the pure-MPI version (*Hint*: the maximum benefit will come when you use OpenMP in the `independent_update()` function in the source file because this contains the bulk of computation).

4. We have used the MPI contiguous and Vector datatype instead of packing the data in a simple buffer for any run-time optimization but you can try changing this to a 1-D buffer to evaluate its effect on timing.

5. Try to evaluate if overlapping of communication and computation yields any advantages in terms of the running time & in general, if it is possible to optimize the program.

6. Change the 5-pt stencil (in 2-D) to a Red-Black Gauss-Seidel that uses the recent-most value of the solution.

7. Extend the program to 3-D (use sub-arrays to implement passing of data or halos/ghost regions). You can also try to implement the 19-pt and 27-pt stencil in 3-D. Remember that with the 27-pt stencil, each process will receive and send data to 26 neighboring processes.

8. If you are able to implement the problem in 3-D, then try to extend this to a Geometric Multigrid using a factor of 2 as the coarsening ratio, and using full restriction and trilinear interpolation.

[2] Initially all boundaries are fixed at $\phi(x, y) = 1$