Università della Svizzera italiana | **Institute of Computing CI**

High-Performance Computing Lab **Institute of Computing**

Student: Jonatan Bella Discussed with: None

Solution for Project 7

1. HPC Mathematical Software for Extreme-Scale Science [85 points]

1. Discretization and Derivation of the Linear System

We can write the given problem as:

$$-\Delta u = f \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega \tag{1}$$

where $f = 20$ (constant) and $\Omega = (0,1) \times (0,1)$

Now, we need to think on how to discretize the problem. Considering a uniform grid on the unit square Ω. x_1 and x_2 directions with spacing $h = \frac{1}{n-1}$, where n is the number of points in each direction:

$$x_i = ih, \quad y_j = jh, \quad i, j = 0, 1, \ldots, n-1 \tag{2}$$

so that we have interior grid points at (x_i, y_j) for $i, j = 1, \ldots, n-2$

The boundary nodes are at $i = 0$, $i = n-1$, $j = 0$, and $j = n-1$, on which u is known to be zero.

Laplacian

The Laplacian in two dimensions is:

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}. \tag{3}$$

Using second-order centered finite differences with $h = \frac{1}{n-1}$:

$$\frac{\partial^2 u}{\partial x_1^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2},$$
$$\frac{\partial^2 u}{\partial x_2^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}.$$

At an interior point (i, j), for $-\Delta u = f$ we have:

$$-\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\right) = f. \tag{4}$$

$$\frac{4u_{i,j} - (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})}{h^2} = f. \tag{5}$$

We can now write the system $Au = b$ as:

- For interior points:
  - Diagonal entries of A: $\frac{4}{h^2}$
  - Adjacent neighbor entries of A: $-\frac{1}{h^2}$
- For the right-hand side b: at interior points, $b_{ij} = f$
- For boundary points: set $A_{ij,ij} = 1$ and $b_{ij} = 0$ to enforce $u = 0$ on the boundary.

1.1. Boundary problem above in Python [25 points]

For the implementation in python. With respect to *ComputeRHS* function, I first define the *rhs* matrix that contains the values of the right-hand side of the equation (forcing function f in this case plus 0 on the boundaries).

Then, on *ComputeMatrix* function, I define the respective differentials and iterate over the grid on both x and y directions. Such that i first check for the boundary cases where, as i explained before in the mathematical derivation, we set to 1 the diagonal such that $A_{row,row} = 1$ and $b_{row} = 0$ to enforce $u = 0$ on the boundary where it means that $1 * u_{i,j} + 0 * (...) = u_{i,j} = 0$

For the interior points, I define the diagonal first where $diag\_val = 2.0 * (1.0/hx2 + 1.0/hy2)$ to account for not uniform grid spacing (different hx and hy), and then I populate the *data* array with the values of the matrix A and the respective row and column indices. Then, I populate the left, right, bottom and top neighbors with a similar approach, as for example, the left neighbor for a given interior point will have as entry $-1.0/hx2$ and the row and column indices will be row and $row - 1$ respectively, given that is the left neighbor. This example applies in an analogous way to the right, bottom and top neighbors.

1.2. Boundary problem above in PETSc [25 points]

The PETSc implementation is similar to the one done in python. For the construction of *ComputeRHS* function: I first make use of the distributed array (DMDA) that manages the parallel decomposition of the grid to get the local grid size (info.xs, info.ys, info.xm, info.ym) which tells us which portion of the global grid this process is responsible for. After computing the grid spacing hx and hy, we loop over all local grid points such that for boundary points (where i == 0, i == mx-1, j == 0, or j == my-1), we set the RHS value to 0.0, since these points are fixed by the Dirichlet boundary condition (u=0) and for interior points, we set the RHS to the constant forcing f = user-¿c. After filling in these local values, we call PETSc routines to put together the global RHS vector.

For the construction of $ComputeMatrix$ function: In the $ComputeMatrix$ function, we again use the DMDA to get information about the local portion of the grid. We also compute the grid spacing hx, hy and their squares (hx2, hy2). we then loop over each local grid point and determine if it's a boundary or interior point such that if it is a boundary point we enforce u=0 by setting a row in the matrix that looks like $A[row, row] = 1$ and no other nonzero entries in that row. (like in the python code) For the interior points, we apply the 5-point stencil for $-\Delta u$. We set the diagonal entry to 2 * $(1/hx^2$ + $1/hy^2)$ and each neighbor (left, right, up, down) to -$1/hx^2$ or -$1/hy^2$, depending on the direction. This mirrors the mathematics we discussed: the diagonal represents the contribution from $u_{i,j}$, and the neighbors represent $u_{i\pm1,j}$ and $u_{i,j\pm1}$.

1.3. Validate and Visualize [10 points]

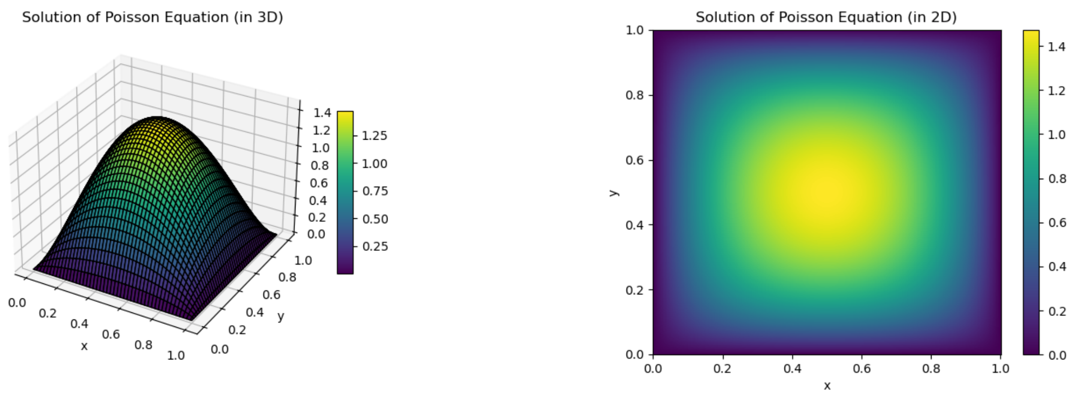

Figure 1: Solution of the Poisson equation using PETSc

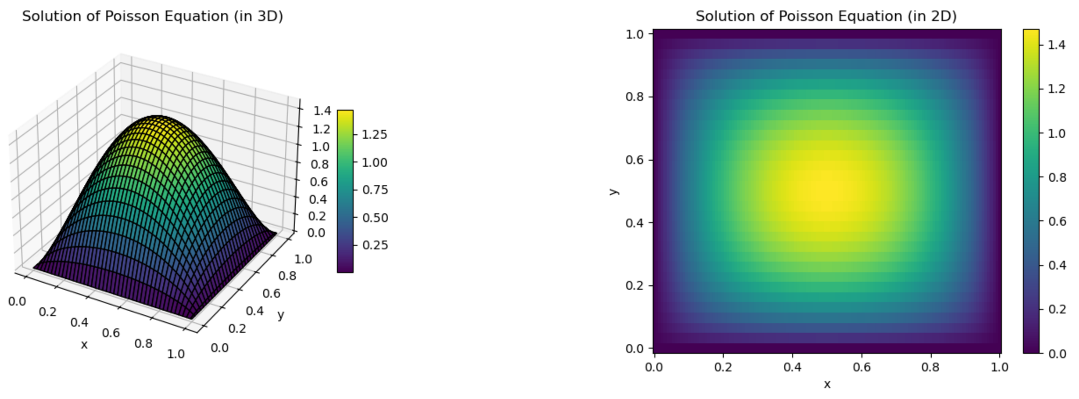

Figure 2: Solution of the Poisson equation using Python Numpy dense solver solution plotted in 3D and 2D

1.4. Performance Benchmark [15 points]

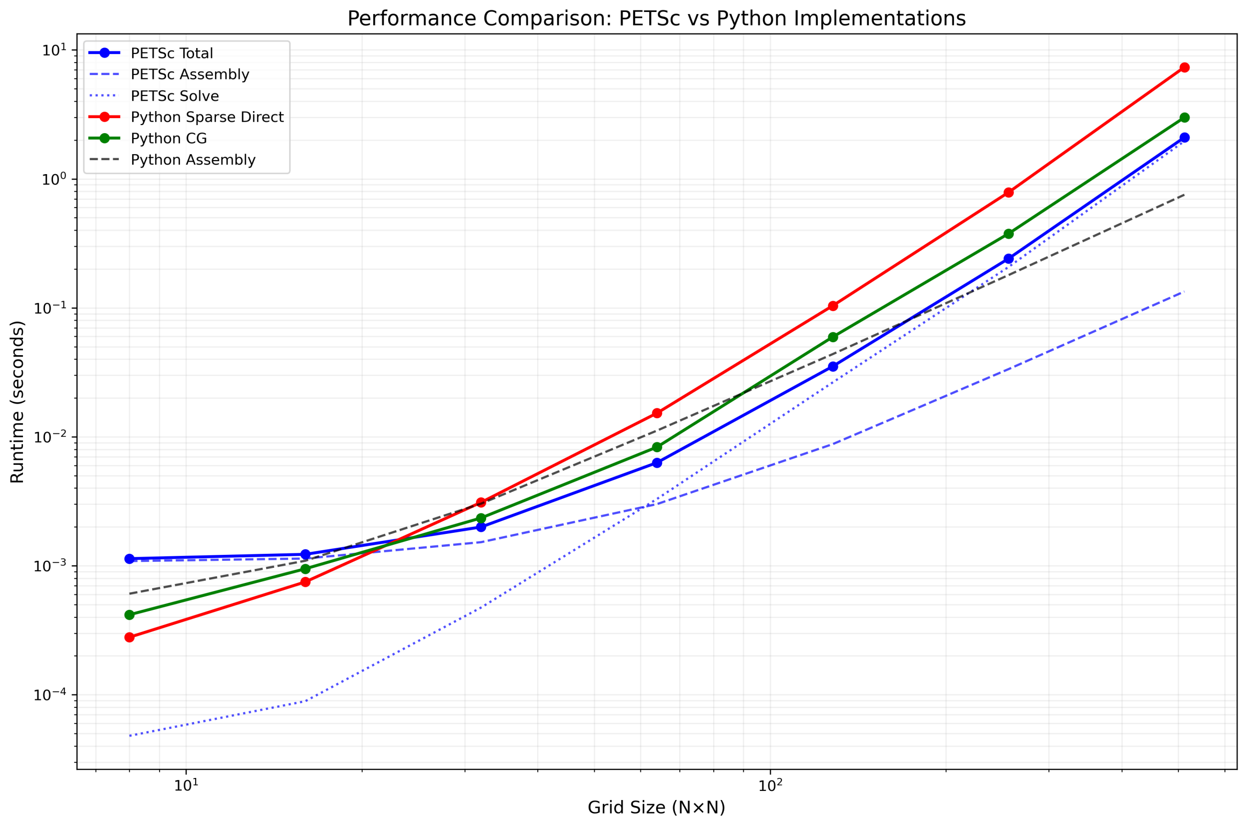

Figure 3: Performance benchmark of the Poisson equation: Python vs PETSc

The performance comparison between both implementations across different grid sizes (8x8 to 512x512) shows that PETSc consistently outperforms the Python implementation. For smaller grids, both implementations show similar performance, but as the grid size increases, PETSc advantage becomes more pronounced.

1.5. Strong Scaling [10 points]

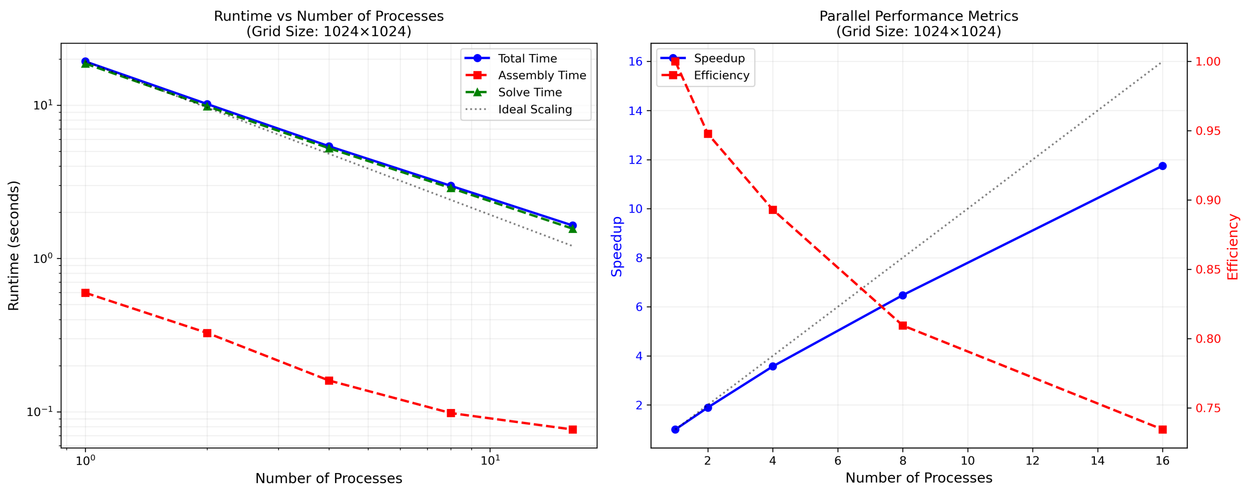

Figure 4: Strong scaling of the Poisson equation: Python vs PETSc

The strong scaling test [1] demonstrates good parallel efficiency for the PETSc implementation. The total runtime decreases from 19.29s (1 process) to 1.64s (16 processes), showing an approximate speedup of 11.76x. While not achieving perfect linear speedup (16×), this is reasonable due to communication overhead. The assembly phase shows better scaling efficiency than the solve phase.

[1] fixed problem size 1024×1024 with increasing process count from 1 to 16