Challenge 3.
A matrix–free parallel solver for the Laplace equation

Luca Formaggia, Alberto Artoni, Beatrice Crippa

AA 23-24

1 Introduction

Consider the Laplace equation

$$\begin{cases} -\Delta u = f(x), & \text{in } \Omega = (0,1)^2, \\ u = 0, & \text{on } \{x = 0\}, \\ u = 0, & \text{on } \{x = 1\}, \\ u = 0, & \text{on } \{y = 0\}, \\ u = 0, & \text{on } \{y = 1\}, \end{cases} \tag{1}$$

modelling the heat diffusion over a square domain with a prescribed temperature (Dirichlet conditions) on the whole boundary.
A possible approach to solve this problem is the so called *Jacobi iteration* method: given a uniform Cartesian decomposition of Ω consisting of n points along each coordinate direction, the goal is to find the discrete solution $u_{ij} = u(x_i, y_j),\ i, j = 1, \ldots, n$ at each point of such Cartesian grid.
We aim at representing the solution as a (dense) matrix U of size $n \times n$; the matrix is initialized with zeroes, except for the first and last rows and columns, which contain the boundary condition values defined in eq. (1).
The algorithm consists of the following iterative procedure.
For $k = 1, \ldots$ until convergence:

1. update each *internal* entries as the average of the values of a four–point stencil:

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

   $\forall i, j = 2, \ldots, n-1$.

2. compute the error as the norm of the increment between $U^{(k+1)}$ and $U^{(k)}$ as follows:

   $$e = \sqrt{h \sum_{i,j} (U^{(k+1)}(i,j) - U^{(k)}(i,j))^2}.$$

   If the error is smaller than a prescribed tolerance, stop.

We have indicated with h the mesh spacing assuming an equal spacing in x and y, i.e. $h = 1/(n-1)$, while $f(i,j) = f(x_i, y_j)$.

2 The challenge

In this challenge you are required to implement a parallel solver for the above problem for a generic f and solve the problem with $f(x) = 8\pi^2 \sin(\pi x)\cos(\pi y)$, whose exact solution is $u(x, y) = \sin(2\pi x)\cos(2\pi y)$.
In particular:

- The number n, f and the number of parallel tasks should be given by the user;
- The nodes are split into blocks of rows, as shown in figure, and assigned to different `MPI` processes;
- The number of rows owned by each processor should be *balanced*, *i.e.* it should be as constant as possible among the different `MPI` ranks; (see figure 1).

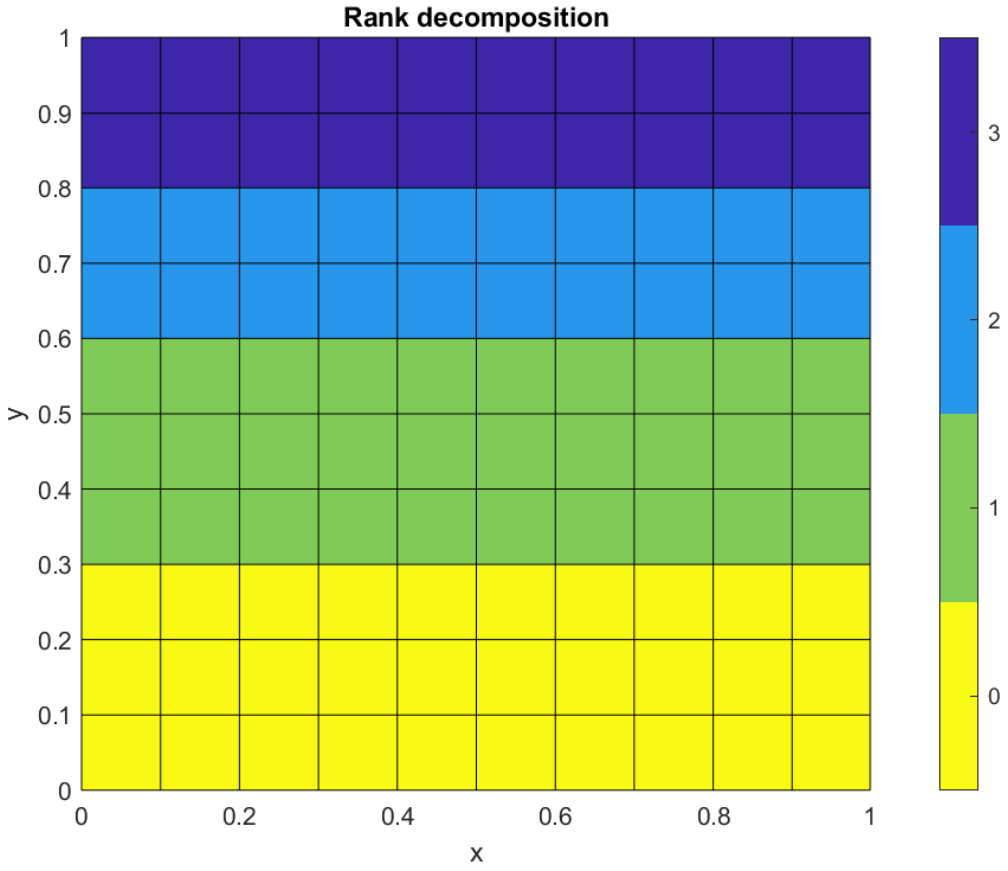

Figure 1: Example of a grid with $n = 11$ whose rows are distributed in parallel among 4 processes.

The tricky part is that the update formula requires access to values of U that are updated by other processors. When we have to update a row at the boundary between two `MPI` ranks, data from adjacent entries must be properly exchanged.
For this challenge you are required to implement a `C++` parallel code in which:

1. Equation (1) is solved with the Jacobi iteration algorithm, by invoking proper `MPI` functions to communicate data between adjacent processors;

2. Insert a proper `OpenMP` directive to further parallelize the local computations (hybrid parallelism);

3. Each processor checks its own local convergence criterion; then the local information is exchanged between all ranks (convergence is reached if **all** ranks satisfy the stopping criterion);

4. The solution and the grid coordinates are exported to file(s) for visualization; this task can be achieved either by letting the *master* rank collect all the local solutions and print them. Export the solution in `vtk` format in such a way it can be opened with ParaView.

Test and plot the performance of the code (serial vs. parallel) as the grid size increases ($n = 2^k$, $k = 4, \ldots, 8$). Compare with the given exact solution in the L_2 norm

$$\sqrt{h \sum_{i,j} (f(i,j) - u(x_i, y_j))^2}.$$

and print the results as function of n (or of h).
Remark: the Jacobi algorithm (without any preconditioner) converges for this problem, but at a (very) slow rate, particularly when n is large, so please choose tolerances wisely and set a large maximum number of iterations.
You are free to choose the code design and which data structures to use. The only mandatory requirement is that the algorithm should **efficiently** run in parallel!

General rules

- You create a GitHub repository, or use the one already set for the previous challenges. *Ensure that you add all teachers of the course as collaborators.*

- The files you put in the git repo should include header and source files, a Makefile, a `README.md` file with a description. Recall *not* to put executables, binary libraries or object files. Those should be regenerated by the Makefile.

- If you refer to stuff in the `Examples/` directory of the course, use the environmental variable `PACS_ROOT` to indicate it in the Makefile, to simplify things.

- Deliver a `test` folder. The `test` folder contains a *script* that runs a small scalability test (with 1,2, 4 cores for instance) and shows the performance of your code. Provide a `README.md` with the instruction to reproduce your results.

- Inside the `test` folder you can have a `data` folder where you can collect some of your results.

- It could be interesting also to report your hardware in a file `hw.info`: `cat /proc/cpuinfo`.

Evaluation criteria

The evaluation will be based on the following requirements:

- Use of GitHub;
- README file, comments in the code, documentation;
- Quality of the code;
- Jacobi solver implementation;
- Scalability performance and discussion on the results;
- Quality of the reproducibility of the result in the `test` folder.
- Hybrid MPI/OpenMP implementation;
- Extras:
    1. Set also the Dirichlet boundary condition through user defined functions and make a test.