Poisson's equation and MPI

In this lab you are going to use a simple parallel algorithm to solve Poisson's equation on the unit square, using Dirichlet boundary conditions.

The description of this lab is fairly lengthy since not all participants, in the HPC-course, are so familiar with partial differential equations, PDEs.

The problem:
Define the computational domain (the unit square) as the following set of points:

$$D = \{(x, y) \,|\, 0 < x, y < 1\}$$

The set of points on the boundary of D (the "edges" of the square) is denoted ∂D. You are given functions $f(x, y)$ and $g(x, y)$. Our PDE-problem can then be formulated in the following way: Find (approximate) $u(x, y)$ such that:

$$\begin{aligned} u_{xx}(x, y) + u_{yy}(x, y) &= f(x, y), \qquad (x, y) \in D \\ u(x, y) &= g(x, y), \qquad (x, y) \in \partial D \end{aligned}$$

Example:
Let $f(x, y) = 2(\cos(x + y) - (1 + x)\sin(x + y))$ and $g(x, y) = (x + 1)\sin(x + y)$, then
$u(x, y) = (x + 1)\sin(x + y)$ solves the above problem. Let us check the answer by computing the derivatives:

$$u_{xx} = 2\cos(x + y) - (1 + x)\sin(x + y) \quad \text{and} \quad u_{yy} = -(1 + x)\sin(x + y) \quad \text{so}$$

$$u_{xx} + u_{yy} = 2(\cos(x + y) - (1 + x)\sin(x + y)) = f(x, y)$$

u satisfies the boundary condition, as well, since $g = u$.
You should use this example in the lab to debug your program, but your program should be written so that it can cope with any reasonable functions f and g.

The serial algorithm: we discretize the unit square using the meshpoints, (x_j, y_k), $j = 0, 1, \ldots, n + 1$, $k = 0, 1, \ldots, n + 1$ where:

$$0 = x_0 < x_1 < ... < x_n < x_{n+1} = 1, \quad 0 = y_0 < y_1 < ... < y_n < y_{n+1} = 1$$

We let the points be equidistant, with spacing $h = 1/(n + 1)$. So

$$x_k = y_k = kh, k = 0, 1, ..., n + 1.$$

Let $u_{j,k}$ be the approximation of $u(x_j, y_k)$. We use standard second order approximations of the second derivatives:

$$u_{xx}(x_j, y_k) \approx \frac{u_{j-1,k} - 2u_{j,k} + u_{j+1,k}}{h^2}, \quad u_{yy}(x_j, y_k) \approx \frac{u_{j,k-1} - 2u_{j,k} + u_{j,k+1}}{h^2}$$

This gives a system of n^2 linear equations in n^2 unknowns ($u_{j,k}$, $1 \le j, k \le n$):

$$\frac{u_{j-1,k} + u_{j+1,k} + u_{j,k-1} + u_{j,k+1} - 4u_{j,k}}{h^2} = f(x_j, y_k), \quad 1 \le j, k \le n$$

Note that the boundary values (from the g-function) will be used when j or k equals 0 or $n + 1$ (e.g. $u_{0,0} = g(x_0, y_0) = g(0, 0)$).

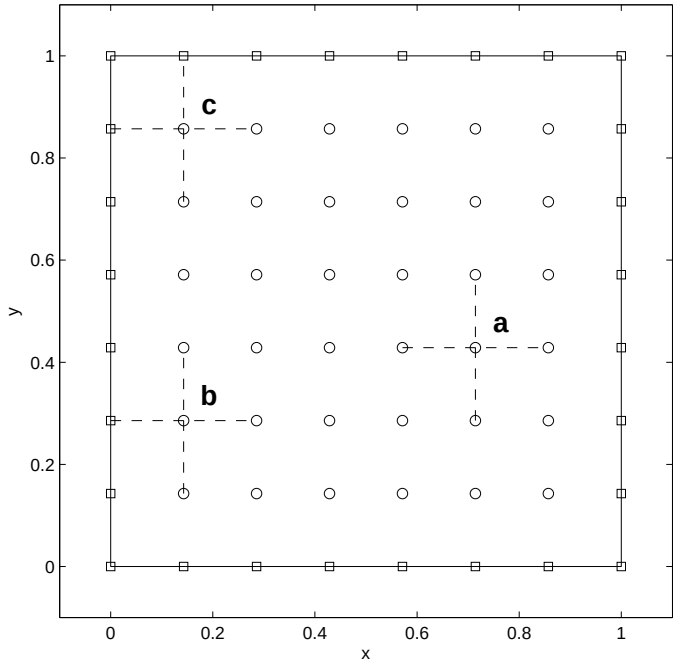
This part of the algorithm is quite realistic, but the following iterative method (for the linear system) is not. We are going to use a very simple iterative method, Jacobi's method, to solve the linear system. One would

probably have used a direct method instead, and even if one would have used an iterative method there are much more effective available.
The main advantage of Jacobi's method is its simplicity and the ease by which it can be parallelized. The big drawback is its slow convergence (which becomes slower with increasing n). This is not a major issue in this lab, since you are not learning how to solve PDEs but how to use MPI. The parallelization issues that appear in the lab are typical and realistic.

One iteration of Jacobi's method can be written like follows. We introduce temporary quantities, $\tilde{u}_{j,k}$, $1 \leq j,k \leq n$, which are computed using two loops over j and k:

$$\tilde{u}_{j,k} = \frac{u_{j-1,k} + u_{j+1,k} + u_{j,k-1} + u_{j,k+1} - h^2 f(x_j, y_k)}{4}, \quad 1 \leq j,k \leq n$$

where the g-values are used for the boundary, as mentioned above. In the image below, internal nodes are marked by circles and boundary nodes by small squares. To form the $\tilde{u}_{j,k}$-value marked by **a**, we use $h^2 f(x_j, y_k)$ and an average of the four u-neighbours, above, below, left of and right of $\tilde{u}_{j,k}$ (the dashed line show the connections). The **b**-node uses a boundary value and the **c**-node uses two boundary values.

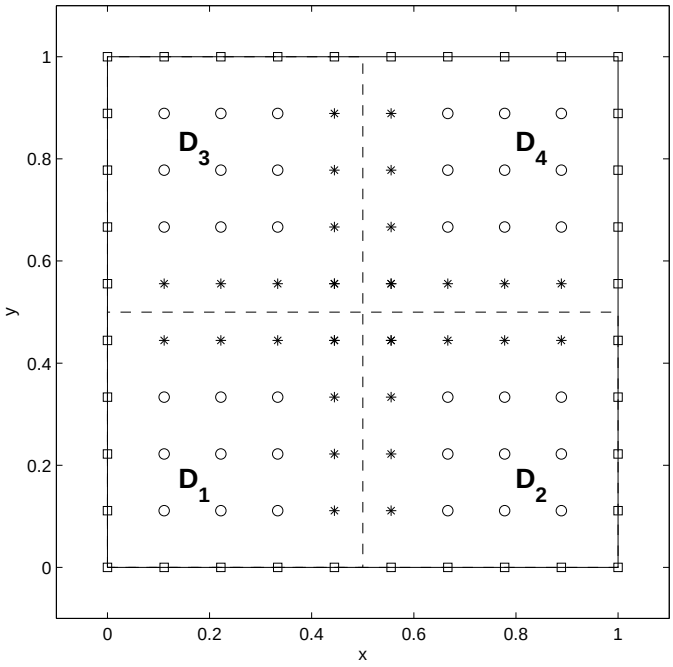


When all $\tilde{u}_{j,k}$-values have been computed we find out the difference between the two successive approximations to the solution, by computing:

$$\delta = \max_{1 \leq j,\, k \leq n} |\tilde{u}_{j,k} - u_{j,k}|$$

Finally we replace $u_{j,k}$ by the temporary values $\tilde{u}_{j,k}$, i.e $u_{j,k} = \tilde{u}_{j,k}$, $1 \leq j,k \leq n$. The value of δ is used to terminate the Jacobi iteration. If $\delta > \tau$ where τ is a tolerance given by the user of the program (you, the reader, in other words), the Jacobi-step is repeated. The iterations are continued until $\delta \leq \tau$.
One can avoid having all the extra $\tilde{u}_{j,k}$-values (there are n^2 of them). It is sufficient to have an extra vector of length n and some scalar variables. This way $u_{j,k}$ can be overwritten by the new values and δ can be computed in the updating loop. Your code **should use** this more efficient implementation.
Note that setting $u_{j,k} = \tilde{u}_{j,k}$, as soon as it has been computed, gives you a different algorithm which is sequential in nature and much harder to parallelize, so don't do that.

Parallelization:
We are going to parallelize the algorithm using so called *domain decomposition*, i.e. we are going to divide the domain, D, into subdomains. Assume you have *four* CPUs. We divide the unit square in four smaller squares. Here is an example. Suppose n is an *even* number. This way no x_j or y_k will equal 0.5.
In the following image $n = 8$. The internal meshpoints are marked by circles and $*$. Boundary points are marked by small squares. We divide D in to the four squares (separated by the dashed lines) marked D_1, D_2, D_3 and D_4. Each CPU is responsible for a subdomain and stores data related with that domain.



CPUs can compute the $u_{j,k}$-values corresponding the circles, in parallel. In order to compute the $*$-values, CPUs must first have communicated and exchanged some data with neighbouring domains. To make this efficient, we want to minimize the number of matrix-elements that are communicated, and when we have to communicate we want to send as much data as possible in one go.

The assignment:
write a parallel MPI-routine, as outlined above, in C or Fortran. To make the lab more realistic I have listed some constraints which your program must satisfy.

- Use four CPUs and the domain decomposition above. Use one master and three slaves (so the master is taking care of one subdomain). The computation should be done in double precision. You decide how the subdomains should be mapped to the processes (which D_j corresponds to which process).
- The master should read the value of n (you may assume that it is an even number) and the tolerance τ. The Jacobi iteration should be terminated when the δ-values (one for each process) **all** are $\leq \tau$.
- You should test your code on the example problem above, and it should compare the computed result with the exact solution, and print the maximum error. The error is composed of three components, roundoff (which is negligible in this application), discretization error (from approximating the derivatives) and the solution error from Jacobi's method. The discretization error is of $\mathcal{O}(h^2)$, so a large n gives a small error. The error in Jacobi's method is controlled by τ. There should be a balance between the two errors. A tiny τ means that discretization error dominates and a large τ gives a large error from the solution of the linear system. How to balance the errors for optimal performance (minimum time) is beyond the scope of this course.

- When you send the code to me I should be able to run it on the test problem without any modifications of your code.
- Each process can use $n^2/2 + 3n + 8$ double precision elements for arrays whose length depend on n. You can use these elements as you please. I used them in the following way. Each process stores a quarter of the f-matrix (the right hand sides), and a quarter of the u-matrix as described in Hint 4 below. Finally I use two vectors with $n/2 + 2$ elements. (I could cope with less in Fortran, but they may be needed in a C-program.) A process can, of course, use scalar variables and arrays having four elements as well.
- Your program may **not** use extra storage for a $\tilde{u}$-matrix, but it should use to more efficient solution mentioned above.
- The $f(x, y)$- and $g(x, y)$-values should be computed *once* and stored.
- The communication should be efficient. You may not make a separate communication call for each `*`-element, for example.

Hints:

1. If you have no previous experience of this kind of PDE-problem, think through the serial case first.
2. Parallel programming is harder than ordinary serial programming and message passing is usually harder then using threads. So, it is important to think about algorithm, datastructures and communication before coding. You will save time by making a plan.
3. If you use Fortran it may be useful to choose the smallest matrix-index to zero (like C). You would also like to use dynamic memory allocation (since the master will be reading the value of n). See the links (for this lab) on www, for more details.
4. In my code I let each process store its own $u_{j,k}$-values as well as neighbouring `*`-values, in the matrix `U(0:n/2+1, 0:n/2+1)`, using Fortran syntax (we assumed that `n` is even). This way I can use the same Jacobi-subroutine (in each process) as in the serial case. It is a slight waste of storage, but quite convenient.
5. Which process (rank) will take care of which subdomain?
6. Think about how (x_j, y_k) will be mapped to each U-matrix. How will row- and column-indices correspond to coordinates?
7. Processes must exchange `*`-data. When should the communication take place? It is inconvenient to have it in the update loop. `MPI_Sendrecv` may be a suitable communication routine.
8. You must think about how to terminate the Jacobi iteration. Different processes will get different δ-values, and your program will not work if just one process, say, ends the iteration.
9. There is a similar, but simpler problem, of computing the maximum error. Each process will have its own value.
10. When sending or receiving a column of a matrix it is not necessary to copy the data to a temporary vector. Say you are passing column number `k` from the matrix `M` to the subroutine `sub`. Since Fortran uses call by reference, `call sub(M(1, k), ...)`, will pass the address of `M(1, k)` to `sub`, which can receive it as a column vector. In Fortran90 `call sub(M(:, k), ...)` is an alternative. If, in Fortran90, you pass a row, `call sub(M(k, :), ...)` the compiler will create a temporary array to store the row.