The Poisson Problem

Background

Partial differential equations play an important role in many branches of science and engineering. Here we consider the Poisson problem which, in two space dimensions \( x \) and \( y \), takes the form

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \quad (x, y) \in \Omega,
\]

where \( u = u(x, y) \) is the function we are seeking, \( f = f(x, y) \) is a source term, and \( \Omega \) is the domain in which we seek the solution. The Poisson equation describes, e.g., the steady state heat distribution in a media with constant heat capacity.

In this assignment we consider the heat distribution in a small square room (ignoring convection and other effects) with a radiator (with a radiation = 200 \(^\circ\) C/m\(^2\)) placed somewhat near the cold wall, and with the temperature kept fixed at the walls: 20\(^\circ\) C at three walls and 0\(^\circ\) C degrees at the fourth wall. Hence, we can take \( \Omega \) as the square

\[
\Omega = \{(x, y) : |x| \leq 1, |y| \leq 1\}
\]

and we have the Dirichlet boundary conditions

\[
\begin{align*}
    u(x, 1) &= 20, & u(x, -1) &= 0, & |x| \leq 1 \\
    u(1, y) &= u(-1, y) = 20, & |y| \leq 1.
\end{align*}
\]

Finally, the radiator is represented by the function

\[
f(x, y) = \begin{cases} 
    200 & , \ 0 \leq x \leq 1/3, \ -2/3 \leq y \leq -1/3 \\
    0 & , \ \text{elsewhere}.
\end{cases}
\]

The problem can be solved by discretization of the problem on a rectangular \( N \times N \) grid, where we represent the solution at grid point \( i, j \) by the value \( u_{i,j} \) (and similarly for \( f \)). The solution can then be computed by repeatedly updating all the inner grid points by means of the finite difference method and the five-point stencil formula

\[
    u_{i,j} \leftarrow \frac{1}{4} \left( u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j} + \Delta^2 f_{i,j} \right),
\]

where \( \Delta \) is the grid spacing. For this problem, the solution on the boundary grid points are given by the boundary conditions, and these values are used when updating the grid points next to the boundary.
The Assignment

1. **Sequential code - Jacobi method**: Write a (sequential) program that solves the discretized problem using the Jacobi update method - preferably implementing the method as a subroutine. Test the program for different values of \( N \), and familiarize yourself with the problem, the solution, and the convergence of the iterations (use the stop criterion discussed in the lecture).

2. **Sequential code - Gauss-Seidel method**: Repeat step 1 by replacing the Jacobi method with the Gauss-Seidel update method. Compare the convergence behaviour between the two methods, i.e. which one converges faster?

   To compare the two methods, you could e.g. compare the number of iterations per second, either based on the whole grid or on a per grid point basis.

3. **OpenMP Jacobi**: Implement a 'simple' OpenMP version of the Jacobi method and report your experiences with speed-up.

   Try to improve the parallelization of this code (think about barriers, the size of parallel regions, etc.). Explain your efforts, the scoping of the variables and why and where you had to introduce new variables to achieve the improvement.

   Investigate how your different OpenMP implementations scale, i.e. measure the speed-up and efficiency (compare to Amdahl’s law). Do this for different numbers of grid points (memory footprint), and try to find the optimal number of threads for a given grid size.

   **Notes (apply to both 3 and 4)**: Use the wall-clock times when comparing different parallel test-runs! Remember also to check if there is enough capacity in terms of free CPUs when conducting your measurements, by comparing the load of the machines (uptime command with the number of CPUs in the machine (cpucount)).

4. **OpenMP Gauss-Seidel**: Compared to the Jacobi method, the Gauss-Seidel method cannot be parallelized in the same 'simple' way as the Jacobi method, without applying a very fine-grained blocking mechanism. Explain why!

   A way around this problem is the introduction of 'temporally blocking', i.e. having the threads working on different iteration steps at the same time. To achieve the same results as in the sequential code, the two conditions below have to be fulfilled. Thread \( t \) can start execution of iteration step \( s \) if

   - thread \( t - 1 \) has already performed step \( s \).
   - thread \( t \) and \( t + 1 \) have performed the same number of steps.
The first condition prevents thread \( t \) from performing step \( s \) before \( t - 1 \), while the second condition prevents \( t \) to perform step \( s + 1 \) before thread \( t + 1 \) performs step \( s \). This will lead to certain overhead in the start-up phase (compare this e.g. to pipelining).

Hints:

- Do **not** evaluate the stop criterion in every iteration step, to avoid too many synchronizations.
- Adapt the sequential version of the Gauss-Seidel version from step 2 to the same mechanism, to have a version to compare your results with.

Again, report your experience with the parallelization and the speed-up.

We emphasize that you shouldn’t necessarily expect a progression from slower to faster performance, when going through this assignment. Rather, the goal is to try different parallelization techniques for the same mathematical problem.