02614 High-Performance Computing - 2019 January 14, 2019

Assignment 2: The Poisson Problem

Your report must be handed in electronically on Campusnet in PDF format!
Please add your source code in a ZIP file!

Deadline: Friday, January 18, 2019 - at midnight!

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

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -f(x,y), \qquad (x,y) \in \Omega,$$

where $u = u(x,y)$ is the function we are seeking, $f = f(x,y)$ is a source term, and Ω 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° C/m^2) placed somewhat near the cold wall, and with the temperature kept fixed at the walls: 20° C at three walls and 0° C degrees at the fourth wall. Hence, we can take Ω as the square

$$\Omega = \{(x,y) : |x| \le 1 \ , \ |y| \le 1\}$$

and we have the Dirichlet boundary conditions

$$u(x,1) = 20, \quad u(x,-1) = 0, \qquad |x| \le 1$$
$$u(1,y) = u(-1,y) = 20, \qquad |y| \le 1.$$

Finally, the radiator is represented by the function

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

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

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

where Δ 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 (see "Notes" below). 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 (sometimes referred to as 'lattice site updates per second').

3. **OpenMP Jacobi:** Implement a 'simple' OpenMP version of the Jacobi method and report your experiences with speed-up. Start really simple, and keep this version, to be able to compare your improvements with this baseline.

   Try to improve the parallelization of this code (think about barriers, the size of parallel regions, data initialization, 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 think about ways to optimize the runtime behaviour (thread placement, etc).

   Does the code scale as expected (discuss)? Compare the scaling behaviour for optimized and non-optimized code (here we refer to compiler optimizations!) - which one scales better? What about the total execution times (wall-clock time)?

4. **Comparison with Mandelbrot:** Compare the scaling behaviour of your OpenMP Jacobi implementation with the scaling behaviour of the Mandelbrot problem, known from the lab exercises. Use problem sizes that are similar, i.e. grid sizes in Jacobi of the same order as the number of pixels in the Mandelbrot problem. Explain your findings!

Notes (for everything above):

- Use the wall-clock times when comparing different parallel test runs!
- Submit your jobs to the batch system (preferred way). Please note down the CPU type used for your experiments in the report.
- Test your parallel implementation, i.e. does it always give the same result, independent of the number of threads.

- Fix the number of iterations to a reasonable number, when doing scaling experiments. There is no need to run until convergence is reached - your time to do experiments is limited! A good estimate for the maximum number of iterations: make it just as large, such that the sequential execution on 1 thread takes at most 5 minutes!
- It might be a good idea to be able to control the 3 main parameters, i.e. grid size N, max. number of iterations $k_{\texttt{max}}$ and the threshold d, either via command line arguments, or to read them from a configuration file.
- Use a modular structure in your program, i.e. use subroutines for most of the tasks. A typical structure of your main program could look like:

  1. get run time parameters (either from command line, e.g. `poisson N k d`, or from a file)
  2. allocate memory for the necessary data fields (S)
  3. initialize the fields with your start and boundary conditions (S)
  4. call iterator (Jacobi or Gauss-Seidel) (S)
  5. print results, e.g. timings, data, etc
  6. de-allocate memory

  (S) above means subroutine.

  To avoid that the compiler does 'tricks' behind your back, e.g. inlining of code, that might change the optimization, etc, it will be a good idea to create a source file for each of the subroutines. In that way, you'll have better control over the code changes and their effects!
- To make your program as flexible as possible, do not fix things like the no. of threads in your code — use the default, i.e. control via the environment variables, like OMP_NUM_THREADS, etc.
- If you run short jobs interactively, remember to check if there is enough capacity in terms of free CPU cores when conducting your measurements, by comparing the load of the machines (`uptime`) command with the number of CPUs in the machine (`cpucount`).

5. **OpenMP Gauss-Seidel:** Compared to the Jacobi method, the Gauss-Seidel method cannot be parallelized in the same 'simple' way as the Jacobi method. Explain why!

   However, there exist ways to parallelize the Gauss-Seidel method. Do a literature search and find methods how one can parallelize the GS method for multi-core systems. Please provide the references (URLs or Journal reference), and give a short summary in your own words how the methods works.

   Note: There is no need to write code for this part of the assignment!