02614 High-Performance Computing - 2020 January 14, 2020

Assignment 2: The Poisson Problem

Your report must be handed in electronically on DTU Inside in PDF format!
Please add your source code in a ZIP file!
Remember the Addendum with the responsibilities!

Deadline: Friday, January 17, 2020 - 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 three space dimensions x, y and z, takes the form

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

where $u = u(x, y, z)$ is the function we are seeking, $f = f(x, y, z)$ 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 cubic 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 five walls and 0° C degrees at the sixth wall. Hence, we can take Ω as the cube

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

and we have the Dirichlet boundary conditions

$$u(x, 1, z) = 20, \quad u(x, -1, z) = 0, \qquad |x| \leq 1 \ , |z| \leq 1$$

$$u(1, y, z) = u(-1, y, z) = 20, \qquad |y| \leq 1, |z| \leq 1$$

$$u(x, y, -1) = u(x, y, 1) = 20, \qquad |x| \leq 1, |y| \leq 1.$$

Finally, the radiator is represented by the function

$$f(x, y, z) = \begin{cases} 200 & , \ -1 \leq x \leq -3/8, \ -1 \leq y \leq -1/2, \ -2/3 \leq z \leq 0 \\ 0 & , \text{ elsewhere.} \end{cases}$$

The problem can be solved by discretization of the problem on a cubic $N \times N \times N$ grid, where we represent the solution at grid point i, j, k by the value $u_{i,j,k}$ (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 seven-point stencil formula

$$u_{i,j,k} \leftarrow \frac{1}{6}\left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} + \Delta^2 f_{i,j,k}\right),$$

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. **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. One solution to the problem is the so-called 'red-black' (or 'coloring') method, where one subdivides the grid into two colored sub-grids, e.g. a 'red' and 'black' one, thus that every 'red' grid point has only 'black' grid points as neighbours, and vice-versa. During the iterations, the 'red' and 'black' grids are then updated in a leap-frog way.

Another approach are 'wavefront' or 'temporal blocking' algorithms. The 'temporally blocked Gauss-Seidel method' is e.g. described in the article by "Wallin et al.: *Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors*" (a PDF copy of the paper is provided on DTU Inside). The paper describes the basic ideas behind the method, and why it is good for cache-based multi-core systems.

With the release of OpenMP 4.5, an extension to the 'ordered' keyword for loops has been introduced, that allows to implement this kind of parallelism. In OpenMP this is called 'doacross'-loops. You can find a description and the principles behind this in Chapter 2 (section 2.4.4) of the book "OpenMP - The Next Step" (this chapter is available as PDF on DTU Inside).

Your task: implement an OpenMP parallel version of the Gauss-Seidel method, using 'doacross'-loops. To make things easier, you can omit the calculation of the stop criterion, and run with max. iterations as the only control parameter.

Investigate how the OpenMP parallel version scales with respect to the number of threads. Explain your findings!

Note: Unfortunately, the GCC implementation of 'doacross'-loops in OpenMP does not work. To do this part of the assignment, you will have to use the 'clang' compiler: `'module load clang/9.2.0'`. The compiler options of this compiler are compatible with the GCC options, so all you need to change, e.g. in your Makefile, is the CC macro, from 'gcc' to 'clang'. Don't expect to get the same runtimes as with GCC, but that is not important here, as we are interested in the scaling!

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 3-4 minutes!

- 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 from the command line, e.g. your program should be called like this: `poisson N k d T ...`. A template can be found on DTU Inside.
  2. allocate memory for the necessary data fields (S) — we provide a routine for the allocation of 3D data structures!
  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!

- For visualization purposes, we provide a subroutine that can dump a file in VTK format, that can be read by tools like ParaView. How to access ParaView on the cluster is described in the README.paraview file, that is part of the material made available on DTU Inside.

- We provide also a subroutine that dumps the 3D grid as a binary, that can then be read by MATLAB, or Python, if you want to use those tools to visualize your data.

- 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`).