Spring 2017: Advanced Topics in Numerical Analysis: High Performance Computing
Assignment 2 (due March 23, 2017)

Handing in your homework: Please create a Git repository on Github, Bitbucket or Gitlab. If you choose your repository to be private, please give me and Bill read access to the repo (our user names are `georgst` and `yuanxunbao`). The repository should contain the C-files, as well as a Makefile. My OpenMP-capable compiler is simply called `gcc`[1]. To hand in your homework, you can either email Bill and me the location of your repo which contains a short `.txt` or LaTeX file that answers the questions/reports timings from this assignment, or you hand in a sheet with the answers/timings that also specifies the location of your repo. To check if your code runs, we will type the following commands[2]:

```
git clone YOURPATH/YOURREPO.git
cd YOURREPO
make
./omp_solved2
...
./omp_solved6
./jacobi2D-omp
./gs2D-omp
```

The git repository https://github.com/NYU-HPC17/homework2.git contains the buggy examples needed for this homework.[3] For an overview over OpenMP, you can use the material and examples from class and the official documentation for the OpenMP Standard 4.5.[4]

1. **Finding OpenMP bugs.** The above repository contains five OpenMP problems that contain bugs. Try to find these bugs and fix them. Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your repository using the naming convention `omp_solved{2,...}.c`, and provide a Makefile to compile the fixed example problems.

2. **OpenMP version of 2D Jacobi/Gauss-Seidel smoothing.** Implement first a serial and then an OpenMP version of the two-dimensional Jacobi and Gauss-Seidel smoothers. This is similar to the problem on the first homework assignment, but for the unit square domain $\Omega = (0,1) \times (0,1)$. For a given function $f : \Omega \to \mathbb{R}$, we aim to find $u : \Omega \to \mathbb{R}$ such that

$$- \Delta u := -(u_{xx} + u_{yy}) = f \text{ in } \Omega, \qquad (1)$$

---

[1] If you want to change the name of your compiler on your machine, you can create an alias for it in your `.bashrc` or `.cshrc` (or whatever shell you use). On my Mac, for instance, I use: `alias gcc='gcc-mp-6'`

[2] Since we will partially autmize this, make sure this will work for the code you hand in.

[3] And yes, these examples can also be found on the web—but for your own sake, please try to understand what's going on and try to find the bugs.

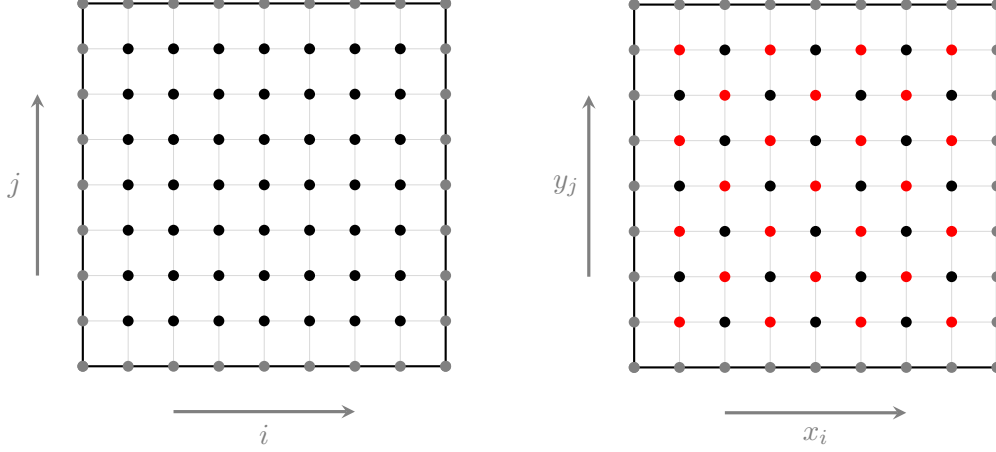[4] http://www.openmp.org/specifications/

Figure 1: Sketch of discretization points for unit square for $N = 7$. Left: Dark points are unknowns, grey points at the boundary are zero. Right: red-black coloring of unknowns. Black and red points can be updated independently in a Gauss-Seidel step.

and $u(x, y) = 0$ for all boundary points $(x, y) \in \partial\Omega := \{(x, y) : x = 0$ or $y = 0$ or $x = 1$ or $y = 1\}$. We go through analogous arguments as in homework 1, where we used finite differences to discretize the one-dimensional version of (1). In two dimensions, we choose the uniformly spaced points $\{(x_i, y_j) = (ih, jh) : i, j = 0, 1, \ldots, N, N+1\} \subset [0, 1] \times [0, 1]$, with $h = 1/(N + 1)$, and approximate $u(x_i, y_j) \approx u_{i,j}$ and $f(x_i, y_j) \approx f_{i,j}$, for $i, j = 0, \ldots, N + 1$; see Figure 1 (left). Using Taylor expansions as in the one-dimensional case results in

$$-\Delta u(x_i, y_j) = \frac{-u(x_i-h, y_j) - u(x_i, y_j-h) + 4u(x_i, y_j) - u(x_i+h, y_j) - u(x_i, y_j+h)}{h^2} + \text{h.o.t.},$$

where h.o.t. stands for a remainder term that is of higher order in h, i.e., becomes small as h is decreased. Hence, we approximate the Laplace operator at a point (x_i, y_j) as follows:

$$-\Delta u_{ij} = \frac{-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1}}{h^2}.$$

This results in a linear system, that can again be written as $A\boldsymbol{u} = \boldsymbol{f}$, where

$$\boldsymbol{u} = (u_{1,1}, u_{1,2}, \ldots, u_{1,N}, u_{2,1}, u_{2,2}, \ldots, u_{N,N-1}, u_{N,N})^\top,$$
$$\boldsymbol{f} = (f_{1,1}, f_{1,2}, \ldots, f_{1,N}, f_{2,1}, f_{2,2}, \ldots, f_{N,N-1}, f_{N,N})^\top.$$

Note that the points at the boundaries are not included, as we know that their values must be zero. Similarly to the one-dimensional case, the resulting Jacobi update for solving this linear system is

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

and the Gauss-Seidel update is given by

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

2

where it depens on the order of the unknowns which entries on the right hand side are based on the kth and which on the $(k+1)$st iteration. Usually, as in the one-dimensional case, one use a single vector $\boldsymbol{u}$ of unknows, which are overwritten and the latest available values are used.

As can be seen, the update at the (i, j)th point in the Gauss-Seidel smoother depends on previously updated points. This dependence makes it difficult to parallelize the Gauss-Seidel algorithm. As a remedy, we consider a variant of Gauss-Seidel, which uses *red-black coloring* of the unknowns. This amounts to "coloring" unknowns as shown in Figure 1 (right), and into splitting each Gauss-Seidel iteration into two sweeps: first, one updates all black and then all the red points (using the already updated red points). The point updates in the red and black sweeps are independent from each other and can be parallelized using OpenMP.[5]

- Write OpenMP implementations of the Jacobi and the Gauss-Seidel method with red-black coloring, and call them `jacobi2D-omp.c` and `gs2D-omp.c`. Make sure your OpenMP codes also compile without OpenMP compilers using preprocessor commands (`#ifdef _OPENMP`) as shown in class.

- Choose the right hand side $f(x, y) \equiv 1$, and report timings for different values of N and different numbers of threads, specifying the machine you run on. These timings should be for a fixed number of iterations as, similar to the 1D case, the convergence is slow, and slows down even further as N becomes larger.

---

[5]Depending on the discretization and the dimension of the problem, one might require more than two colors to ensure that updates become independent from each other and allow for parallelism. Efficient coloring for unstructured meshes with as little colors as possible is a difficult research question.