Spring 2023: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 3, due Apr 2, 2023

Handing in your homework: Use a Git link as described in the previous homework assignment. Include a Makefile and a short write-up documenting your results. To test your code, we will run

```
git clone YOURPATH/YOURREPO.git
cd YOURREPO/hw3/
make
./omp-scan
./jacobi2D-omp
./gs2D-omp
```

The git repository https://github.com/NYU-HPC23/homework3.git contains the serial scan function implementation. For an overview over OpenMP, you can use the material and examples from class and the official documentation for the OpenMP Standard 5.0.[1]

1. **OpenMP warm-up.** Consider the following code and assume it is executes by two threads. The for-loops are executed in two chunks, one per thread, and the independent functions `f(i)` take i milliseconds.

```
#pragma omp parallel
{
    #pragma omp for schedule(static)
    for (i = 1; i < n; i++)
        f(i)
    #pragma omp for schedule(static)
    for (i = 1; i < n; i++)
        f(n-i)
}
```

   (a) How long would each threads spend to execute the parallel region? How much of that time would be spent in waiting for the other thread?

   (b) How would the execution time of each thread change if we used `schedule(static,1)` for both loops?

   (c) Would it improve if we used `schedule(dynamic,1)` instead?

   (d) Is there an OpenMP directive that allows to eliminate the waiting time and how much would the threads take when using this clause?

2. **Parallel Scan in OpenMP.** This is an example where the shared memory parallel version of an algorithm requires some thinking beyond parallelizing for-loops. We aim at parallelizing a scan-operation with OpenMP (a serial version is provided in the homework repo). Given a (long) vector/array $\boldsymbol{v} \in \mathbb{R}^n$, a scan outputs another vector/array $\boldsymbol{w} \in \mathbb{R}^n$ of the same size with entries

$$w_k = \sum_{i=1}^{k} v_i \text{ for } k = 1, \ldots, n.$$

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

To parallelize the scan operation with OpenMP using p threads, we split the vector into p parts $[v_{k(j)}, v_{k(j+1)-1}]$, $j = 1, \ldots, p$, where $k(1) = 1$ and $k(p+1) = n+1$ of (approximately) equal length. Now, each thread computes the scan locally and in parallel, neglecting the contributions from the other threads. Every but the first local scan thus computes results that are off by a constant, namely the sums obtains by all the threads with lower number. For instance, all the results obtained by the the r-th thread are off by

$$\sum_{i=1}^{k(r)-1} v_i = s_1 + \cdots + s_{r-1}$$

which can easily be computed as the sum of the partial sums $s_1, \ldots, s_{r-1}$ computed by threads with numbers smaller than r. Computing this correction can be done in serial. The update, where the partial sums are all corrected by the correction should then be done in parallel again.

- Parallelize the provided serial code. Run it with different thread numbers and report the architecture you run it on, the number of cores of the processor and the time it takes.

3. **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

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

and $u(x,y) = 0$ for all boundary points $(x,y) \in \partial\Omega := \{(x,y) : x = 0 \text{ or } y = 0 \text{ or } x = 1 \text{ or } y = 1\}$. 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 to 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

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

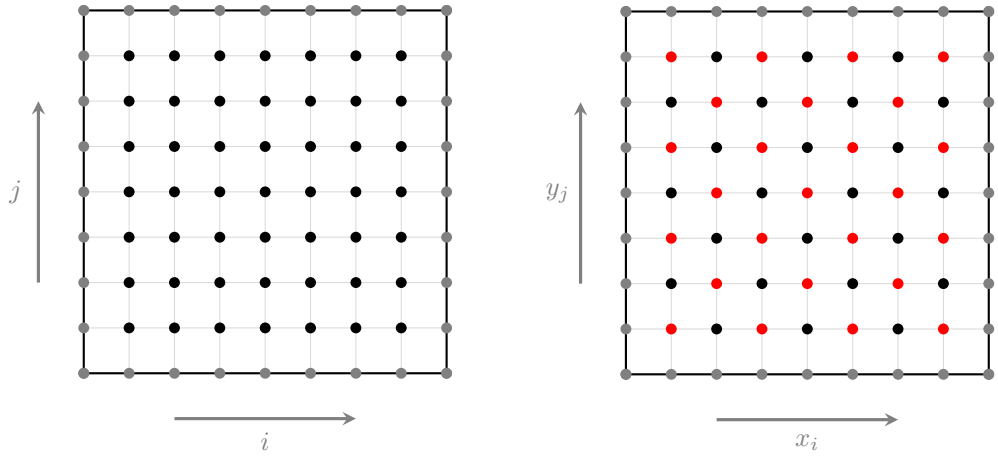

Figure 1: Sketch of discretization points for unit square for $N = 7$. Left: Dark points are unknowns, gray points at the boundary are zero. Right: red-black coloring of unknowns. Black and red points can be updated independently in a Gauss-Seidel step.

where it depends 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. The above update formula is for lexicographic ordering of the points, i.e., we sweep from left to right first and go row by row from the bottom to the top. Usually, as in the one-dimensional case, one use a single vector $\boldsymbol{u}$ of unknowns, 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.[2] To detail the equations, this become the following update, where colors of the unknowns correspond to the colors of points in the figure, i.e., first we update all red points, i.e., (i, j) corresponds to indices for red points,

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

and then we update all black points, i.e., (i, j) are indices corresponding to black points:

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

At the end, every point is on level $(n+1)$ and we repeat.

- Write OpenMP implementations of the Jacobi and the Gauss-Seidel method with red-black coloring, and call them `jacobi2D-omp.cpp` and `gs2D-omp.cpp`. Make sure your OpenMP codes also compile without OpenMP compilers using preprocessor commands (`#ifdef _OPENMP`) as I'll post about on Ed. Note that when implemented correctly, the results for both methods should *not* vary when you change the number of threads.

[2] 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.

- 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.[3]

4. **Preview for final project summary on next assignment.** On the next assignment, you will be asked to summarize your final project, i.e., *what* you will work on and *with whom* (ideally, groups of 2). I'll give a sketch of what is expected from your final project in one of that next classes, and will also provide examples. The main goal of the final project is to show that you are using any of the tools you learned in this class for a real problem which is either from your own research, from a list of topics or something you pitch to me and the TAs. So please start thinking of your final project and communicate with me and/or the TAs about it.

[3] Note again that these smoothers by themselves are slow by they are a crucial part of the multigrid algorithm, which is an optimal complexity method to solve elliptic systems such as (1).