Spring 2022: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 2 (due Mar. 21, 2022)

Handing in your homework: Please create a Git repository on Github to hand in your homework. If you choose your repository to be private, please give Melody and Cai (who are helping with grading) read access to the repo (Melody's username is `melodyshih` and Cai's is `caim-d`). The repository should contain the source files, as well as a Makefile. To hand in your homework, please email me and Melody together a single message with the location of your repo. Generate a `hw2` directory in this repo, which contains all the source code and a short `.txt` or LaTeX file that answers the questions/reports timings from this assignment. Alternatively, you can 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[1]:

```
git clone YOURPATH/YOURREPO.git
cd YOURREPO/hw2/
make
./MMult1
./val_test01_solved
./val_test02_solved
./omp_solved2
...
./omp_solved6
./jacobi2D-omp
./gs2D-omp
```

The folder homework02 in the git repository https://github.com/pehersto/HPCSpring2022 contains the matrix-matrix multiplication code you can start with, and the buggy code examples needed for this homework.[2] For an overview over OpenMP, you can use the material and examples from class and the official documentation for the OpenMP Standard 5.0.[3]

1. **Finding Memory bugs.** The homework repository contains two simple programs that contain bugs. Use valgrind 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 `val_test01_solved.cpp`, `val_test02_solved.cpp`, and use the Makefile to compile the example problems.

2. **Optimizing matrix-matrix multiplication.** In this homework you will optimize the matrix-matrix multiplication code from the last homework using blocking. This increases

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

[2] 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.

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

the computational intensity (i.e., the ratio of flops per access to the slow memory) and thus speed up the implementation substantially. The code you can start with, along with further instructions are in the source file `MMult1.cpp`. Specifying what machine you run on, hand in timings for various matrix sizes obtained with the blocked version and the OpenMP version of the code.

Solution:

- Without open mp enabled

Dimension	Time	Gflop/s	GB/s	Error
16	13.534783	0.147768	2.364282	0.000000e+00
64	13.367776	0.149625	2.394006	0.000000e+00
112	13.221223	0.151319	2.421098	0.000000e+00
160	13.281658	0.151114	2.417819	0.000000e+00
208	13.430287	0.150090	2.401445	0.000000e+00
256	13.375470	0.150519	2.408308	0.000000e+00
304	13.638245	0.148318	2.373093	0.000000e+00
352	13.362954	0.150135	2.402168	0.000000e+00
400	13.700663	0.149482	2.391709	0.000000e+00
448	14.383473	0.150031	2.400499	0.000000e+00
496	14.888363	0.147527	2.360427	0.000000e+00
544	14.999315	0.150263	2.404215	0.000000e+00
592	13.945471	0.148776	2.380411	0.000000e+00
640	14.005553	0.149737	2.395795	0.000000e+00
688	17.708525	0.147120	2.353926	0.000000e+00
736	16.524709	0.144761	2.316172	0.000000e+00
784	19.957219	0.144877	2.318032	0.000000e+00
832	15.913667	0.144764	2.316219	0.000000e+00
880	18.874080	0.144425	2.310799	0.000000e+00
928	22.147929	0.144335	2.309355	0.000000e+00
976	25.856119	0.143829	2.301262	0.000000e+00
1024	14.397250	0.149159	2.386549	0.000000e+00
1072	16.969928	0.145189	2.323027	0.000000e+00
1120	19.355259	0.145173	2.322764	0.000000e+00
1168	22.085118	0.144297	2.308760	0.000000e+00
1216	24.844288	0.144745	2.315923	0.000000e+00
1264	28.064528	0.143917	2.302679	0.000000e+00
1312	31.222667	0.144664	2.314630	0.000000e+00
1360	34.812859	0.144513	2.312209	0.000000e+00
1408	38.139059	0.146375	2.342006	0.000000e+00
1456	42.921286	0.143827	2.301237	0.000000e+00
1504	47.212194	0.144118	2.305894	0.000000e+00
1552	52.014283	0.143742	2.299866	0.000000e+00
1600	56.522795	0.144933	2.318923	0.000000e+00
1648	62.187682	0.143945	2.303124	0.000000e+00
1696	68.265190	0.142925	2.286800	0.000000e+00
1744	74.129680	0.143112	2.289799	0.000000e+00
1792	77.677903	0.148165	2.370645	0.000000e+00
1840	86.903327	0.143366	2.293861	0.000000e+00
1888	93.668724	0.143695	2.299118	0.000000e+00
1936	100.522664	0.144372	2.309947	0.000000e+00
1984	107.902887	0.144751	2.316018	0.000000e+00

- With OpenMP enabled:

Dimension	Time	Gflop/s	GB/s	Error
16	3.474521	0.575620	0.719525	0.000000e+00
64	3.657869	0.546810	0.478459	0.000000e+00
112	3.498952	0.571776	0.469673	0.000000e+00
160	3.578242	0.560901	0.448721	0.000000e+00
208	3.984023	0.505960	0.398930	0.000000e+00
256	18.580066	0.108356	0.084653	0.000000e+00
304	3.525119	0.573825	0.445469	0.000000e+00
352	4.095753	0.489838	0.378511	0.000000e+00
400	4.162914	0.491963	0.378812	0.000000e+00
448	5.930994	0.363846	0.279382	0.000000e+00
496	4.600347	0.477449	0.365787	0.000000e+00
544	5.192025	0.434098	0.331957	0.000000e+00
592	3.811350	0.544360	0.415626	0.000000e+00
640	13.141048	0.159588	0.121686	0.000000e+00
688	4.944496	0.526906	0.401306	0.000000e+00
736	4.950728	0.483187	0.367643	0.000000e+00
784	5.557920	0.520220	0.395473	0.000000e+00
832	6.484944	0.355242	0.269847	0.000000e+00
880	5.675969	0.480251	0.364554	0.000000e+00
928	6.798653	0.470198	0.356702	0.000000e+00
976	8.027142	0.463285	0.351261	0.000000e+00
1024	36.620530	0.058642	0.044439	0.000000e+00
1072	5.341914	0.461230	0.349364	0.000000e+00
1120	6.601534	0.425637	0.322268	0.000000e+00
1168	7.033753	0.453076	0.342911	0.000000e+00
1216	9.082006	0.395958	0.299573	0.000000e+00
1264	7.400298	0.545786	0.412793	0.000000e+00
1312	9.278300	0.486814	0.368079	0.000000e+00
1360	13.093622	0.384226	0.290430	0.000000e+00
1408	42.677493	0.130809	0.098850	0.000000e+00
1456	12.508684	0.493517	0.372850	0.000000e+00
1504	14.855205	0.458031	0.345960	0.000000e+00
1552	18.956051	0.394418	0.297847	0.000000e+00
1600	25.967594	0.315470	0.238180	0.000000e+00
1648	21.823767	0.410178	0.309624	0.000000e+00
1696	24.698791	0.395032	0.298137	0.000000e+00
1744	24.239252	0.437673	0.330263	0.000000e+00
1792	120.051351	0.095869	0.072330	0.000000e+00
1840	23.484675	0.530517	0.400194	0.000000e+00
1888	36.990141	0.363873	0.274447	0.000000e+00
1936	33.935641	0.427652	0.322506	0.000000e+00
1984	44.739503	0.349111	0.263241	0.000000e+00

3. **Finding OpenMP bugs.** The homework repository contains five OpenMP problems that

contain bugs. These files are in C, but they can be compiled with the C++ compiler. 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.

4. **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

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

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. The above update formula is for

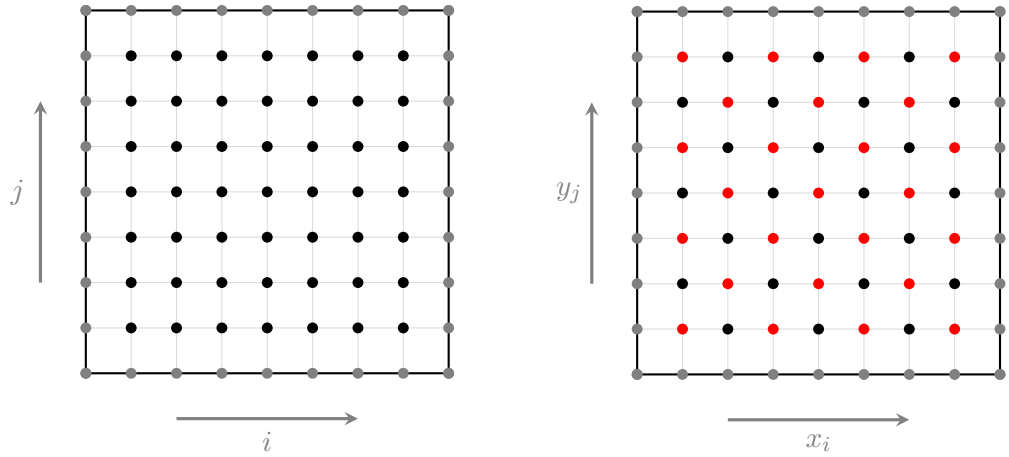

Figure 1: Sketch of discretization points for unit square for $N = 7$. Left: Dark points are unknowns, grey points at the boundary are zero. Right: red-black coloring of unknowns. Black and red points can be updated independently in a Gauss-Seidel step.

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 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.[4] 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 shown in class.

[4] 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.

Solution:

- For jacobi:
  - N = 100, num threads = 4:

```
Maxiter reached (100 iterations)! norm=2.085344e-03
jacobi serial = 0.013429s
Num threads: 4
Max iter count reached (100 iterations). norm=2.085344e-03
jacobi parallel = 0.004404s
```

  - N = 100, num threads = 8:

```
Maxiter reached (100 iterations)! norm=2.085344e-03
jacobi serial = 0.013474s
Num threads: 8
Max iter count reached (100 iterations). norm=2.085344e-03
jacobi parallel = 0.003108s
```

  - N = 1000, num threads = 4:

```
Maxiter reached (1000 iterations)! norm=2.371637e-04
jacobi serial = 14.977925s
Num threads: 4
Max iter count reached (1000 iterations). norm=2.371637e-04
jacobi parallel = 3.555918s
```

  - N = 1000, num threads = 8:

```
Maxiter reached (1000 iterations)! norm=2.371637e-04
jacobi serial = 14.979638s
Num threads: 8
Max iter count reached (1000 iterations). norm=2.371637e-04
jacobi parallel = 1.833424s
```

- For gauss-seidel:
  - N = 100, num threads = 4:

```
Maxiter reached (100 iterations)! norm=3.846189e-03
gauss-seidel (serial) = 0.010911s
Maxiter reached (100 iterations)! norm=3.848755e-03
gauss-seidel colored (serial) = 0.011035s
```

```
Num threads: 4
Maxiter reached (100 iterations)! norm=3.848755e-03
gauss-seidel colored (parallel) = 0.004148s
```

- N = 100, num threads = 8:

```
Maxiter reached (100 iterations)! norm=3.846189e-03
gauss-seidel (serial) = 0.011066s
Maxiter reached (100 iterations)! norm=3.848755e-03
gauss-seidel colored (serial) = 0.011016s
Num threads: 8
Maxiter reached (100 iterations)! norm=3.848755e-03
gauss-seidel colored (parallel) = 0.004015s
```

- N = 1000, num threads = 4:

```
Maxiter reached (1000 iterations)! norm=4.638949e-04
gauss-seidel (serial) = 10.971889s
Maxiter reached (1000 iterations)! norm=4.639037e-04
gauss-seidel colored (serial) = 11.104340s
Num threads: 4
Maxiter reached (1000 iterations)! norm=4.639037e-04
gauss-seidel colored (parallel) = 2.917653s
```

- N = 1000, num threads = 8:

```
Maxiter reached (1000 iterations)! norm=4.638949e-04
gauss-seidel (serial) = 10.917450s
Maxiter reached (1000 iterations)! norm=4.639037e-04
gauss-seidel colored (serial) = 11.069117s
Num threads: 8
Maxiter reached (1000 iterations)! norm=4.639037e-04
gauss-seidel colored (parallel) = 1.488984s
```