Parallel Computer Architecture
Jacobi Equation Solver

Prof. Naga Kandasamy
ECE Department, Drexel University

April 26, 2019

This assignment, worth ten points, is due May 5, 2019, by 11:59 pm. You can work on it in groups of up to two people.

The *Jacobi method*, named after Carl Gustav Jacob Jacobi, is an iterative algorithm to solve a system of linear equations. This technique has many applications in numerical computing. For example, it can be used to solve the following differential equation in two variables x and y:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0. \tag{1}$$

The above equation solves for the steady-state value of a function $f(x, y)$ defined over a physical two-dimensional (2D) space where f is a given physical quantity. In this assignment, f represents the heat as measured over a metal plate shown in Fig. 1, and we wish to solve the following problem: given that we know the temperature at the edges of the plate, what ends up being the steady-state temperature distribution inside the plate?

The 2D space of interest is first discretized via a uniform grid in which Δ is the spacing—for example, in millimeters—between grid points along the two Cartesian dimensions. If Δ is sufficiently small, we can approximate the second-order derivatives in (1) using the Taylor series as

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} &= \frac{f(x+\Delta, y) - 2f(x, y) + f(x-\Delta, y)}{\Delta^2}, \text{ and} \\ \frac{\partial^2 f}{\partial y^2} &= \frac{f(x, y+\Delta) - 2f(x, y) + f(x, y-\Delta)}{\Delta^2}. \end{aligned} \tag{2}$$

Substituting the above equations in (1), we obtain

$$f(x, y) = \frac{1}{4}\left(f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)\right) \tag{3}$$

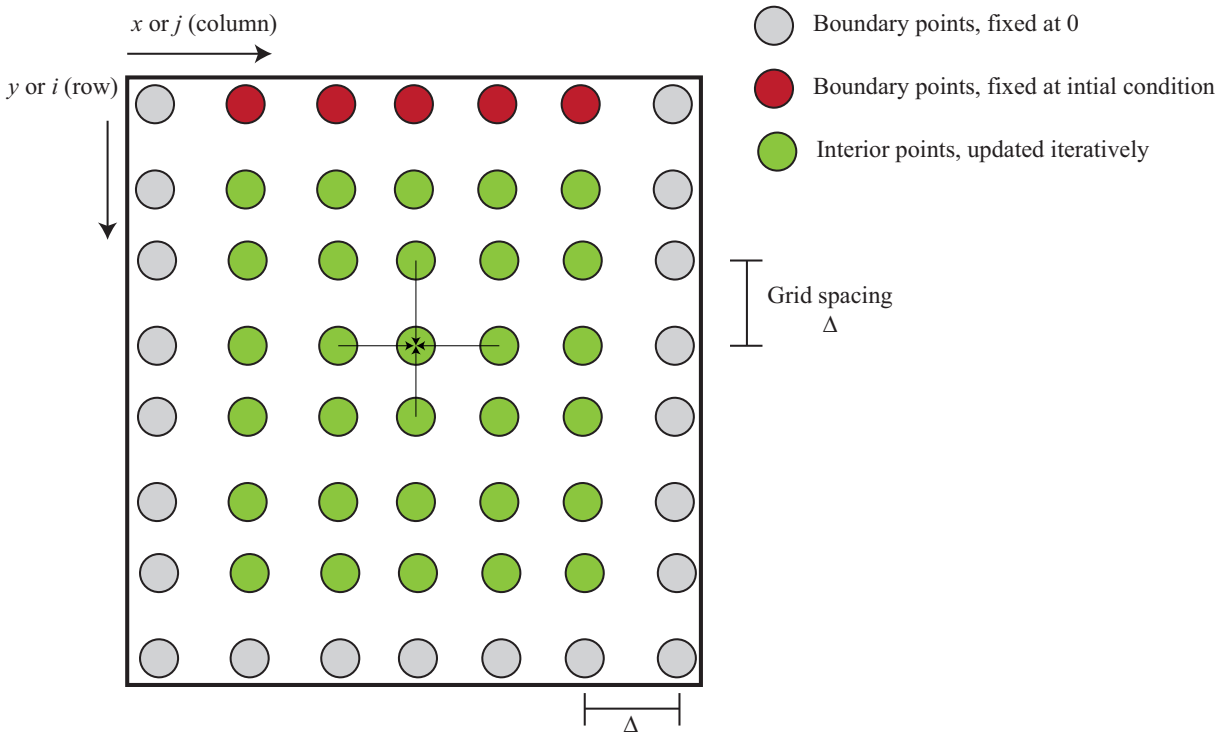

Figure 1: A metal plate with uniformly-spaced grid points. In this example, heat is applied to the row of boundary points colored in red.

The above equation amounts to taking the average of the surrounding four grid points to calculate what the temperature should be for the grid point being evaluated. This is shown graphically in Fig. 1 where the value of each grid point in green is calculated using values from its top, left, right, and down neighbors. Note that we do not calculate values for any of the points the lie on the plate boundary: points at the top are fixed to the value of the heat being applied and the rest are fixed at zero. (Heat could be applied from multiple directions as well.) Since f is simply a temperature value, we can simplify (3) and rewrite it in terms of grid coordinates as

$$u_{i,j} = \frac{1}{4}\left(u_{i,j-1} + u_{i-1,j} + u_{i,j+1} + u_{i+1,j}\right), \tag{4}$$

where $u_{i,j}$ is the temperature at grid location (i, j). We can now solve for at each grid point in iterative fashion. In the sequential implementation shown below, there is a loop-carried dependency in the following sense: since the grid is traversed row-by-row, when updating the value for $u_{i,j}$ in line 15, the most up to date values for $u_{i-1,j}$ (neighbor to the top) and $u_{i,j-1}$ (neighbor to the left) will be used whereas $u_{i,j+1}$ (neighbor to the right) and $u_{i+1,j}$ (neighbor to the bottom) will have values that were computed during the previous iteration.

```c
/* The grid is stored in an n x n array u. */

int done = 0;
int i, j, m;
float old, new, diff;
float eps = 1e-2; /* Convergence criteria. */

while (!done) {
    diff = 0;
    m = 0;

    /* Update each interior grid point. */
    for (i = 1; i < n - 1; i++){ /* Iterate over rows.*/
        for (j = 1; j < n - 1; j++) { /* Iterate over columns. */
            old = u[i][j]; /* Save value from previous iteration. */

            /* Update value for current iteration. */
            new = 0.25*(u[i][j-1] + u[i-1][j] + u[i][j+1] + u[i-1][j]);
            u[i][j] = new;

            /* Accumulate difference between updated and old value.*/
            diff += fabs(new - old);
            m++;
        }
    }
    /* Test for convergence. */
    if (diff/m < eps)
        done = 1;
}
```

The iterative algorithm uses

$$\frac{1}{m}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}|u_{i,j}^{k-1} - u_{i,j}^{k}| < \epsilon, \tag{5}$$

as the convergence test, where m is the number of interior grid points and ϵ is a user-specified small value. The superscript k denotes the iteration number.

In terms of developing the parallel version, the order in which the grid points are updated in the sequential algorithm is not fundamental to the solution; it is simply one possible ordering that is convenient to program sequentially. Since the algorithm is not an exact solution method but rather iterates until convergence, we can update the grid points in a *different order* as long as we use updated values for grid points frequently enough. In the Jacobi method, we don't use updated values from the current iteration for any grid points but always use the values as they were at the end of the previous iteration. That is,

$$u_{i,j}^{k} = \frac{1}{4}\left(u_{i,j-1}^{k-1} + u_{i-1,j}^{k-1} + u_{i,j+1}^{k-1} + u_{i+1,j}^{k-1}\right), \tag{6}$$

Using the provided sequential program as a starting point, develop a parallel version of the Jacobi method. The program accepts arguments for the grid dimension, the number of threads, as well as the temperature range to apply to the north end of the plate. For example, executing the program as

```
$ ./solver 512 4 100 150
```

will create a 512×512 grid in which the elements along the north boundary are initialized to temperature values in the range $[100, 150]$. The program applies the update rule to each element within the grid until the specified convergence criteria is satisfied. The solution provided by the omp implementation using four threads is compared to that generated by the reference code.

Upload all source files needed to run your code as a single zip file on BBLearn. The code must compile and run on the xunil cluster. Also, include a brief report describing: (1) the design of your multi-threaded implementations, using code or pseudocode to clarify the discussion; (2) the speedup obtained over the serial version for 4, 8, and 16 threads, for grid sizes of 512×512, 1024×1024, and 2048×2048 elements. Initialize temperature values to be in the range $[1000, 1500]$ via the command line.