Iterative Jacobi Solver

Prof. Naga Kandasamy
ECE Department, Drexel University

This assignment, worth twenty points, is due May 10, 2023, by 11:59 pm via BBLearn. You may work on this problem in a team of up to two people. One submission per group will suffice. Please submit original work.

Consider the following system of linear equations $Ax = b$ with n equations and n unknowns:

$$\begin{array}{ccccc}
a_{0,0}x_0 & + a_{0,1}x_1 & + \cdots & + a_{0,n-1}x_{n-1} & = b_0, \\
a_{1,0}x_0 & + a_{1,1}x_1 & + \cdots & + a_{1,n-1}x_{n-1} & = b_1, \\
. & . & & . & . \\
a_{i,0}x_0 & + a_{i,1}x_1 & + \cdots & + a_{i,n-1}x_{n-1} & = b_i, \\
. & . & & . & . \\
a_{n-1,0}x_0 & + a_{n-1,1}x_1 & + \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1},
\end{array}$$

where the unknowns are $x_0, x_1, x_2, \ldots, x_{n-1}$. Gaussian elimination is one way to solve for these unknowns. Another method is by iteration. In the above system, the i^{th} equation

$$a_{i,0}x_0 + a_{i,1}x_1 \cdots a_{i,i}x_i \cdots + a_{i,n-1}x_{n-1} = b_i$$

can be rearranged as

$$x_i = \frac{1}{a_{i,i}} \left(b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_1 \cdots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \cdots + a_{i,n-1}x_{n-1})\right),$$

or as

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j}x_j\right) \tag{1}$$

This equation gives x_i in terms of the other unknowns and can be used as an iteration formula for each of the unknowns to obtain better approximations. The iterative method used in the reference code provided to you is the *Jacobi iteration*. It can be shown that the Jacobi method will converge if the diagonal values of a have an absolute value greater than the sum of the absolute values of the other a's on the row; that is, the array of a's is *diagonally dominant*. In other words, convergence is guaranteed if

$$\sum_{j \neq i} |a_{i,j}| \leq |a_{i,i}|.$$

The matrix A generated by the program satisfies the above condition. Note however, that this condition is a sufficient but not a necessary condition; that is, the method may converge even if the array is not diagonally dominant. The iteration formula in (1) is numerically unstable, however, in that it will not work if any of the diagonal elements are zero because it would require dividing by zero — but you do not have to worry about this issue for the assignment. An example of how the iterative method works to solve the system of equations is illustrated using a small example in the file called `jacobi_example.pdf`.

Given arrays `a[][]` and `b[]` holding the constants in the equation, `x[]` holding the unknowns, and a user-defined tolerance for convergence, the sequential code that implements the Jacobi iteration can be developed as follows.

```
for (i = 0; i < n; i++)          /* Initialize unknowns */
    x[i] = b[i];

done = 0;
while (!done) {
    for (i = 0; i < n; i++){ /* Loop iterates over each unknown */
        sum = 0;
        for (j = 0; j < n; j++) { /* Implement Equation 1 */
            if (i != j)
                sum += a[i][j] * x[j];
        }
        new_x[i] = (b[i] - sum)/a[i][i];
    }
    /* Update unknown values and test for convergence */
    ssd = 0;
    for (i = 0; i < n; i++) {
        ssd += (x[i] - new_x[i])^2;
        x[i] = new_x[i];
    }
    if (sqrt(ssd) < TOLERANCE)
        done = 1;
} /* End while. */
```

The convergence criteria tests the square root of the sum of the squared differences (SSD) of the `x` values from two consecutive iterations — the current iteration and the one before — against a user-defined error tolerance as

$$\sqrt{\sum_{i=0}^{n-1}(x_i^k - x_i^{k-1})^2} \leq \text{error tolerance},$$

where k is the iteration number.

The program given to you accepts the width of the square matrix as the command-line parameter. The solution provided by the multi-threaded implementation is compared to that generated by the reference code by printing out the relevant statistics.

Develop parallel formulations of the Jacobi solver using pthreads:

- **(10 points)** Develop a parallel formulation of the solver using the *chunking* design pattern. Complete the *compute_using_pthreads_v1()* function to achieve this functionality. You may develop additional code as needed.
- **(10 points)** Develop a parallel formulation of the solver using the *striding* design pattern. Complete the *compute_using_pthreads_v2()* function to achieve this functionality. You may develop additional code as needed.
- Upload all source files needed to run your code on xunil as a single zip file on BBLearn. Also, provide a short report describing the parallelization process, using code or pseudocode to help the discussion, and the speedup obtained over the serial version for matrix sizes of 512×512, 1024×1024, and 2048×2048, for 4, 8, 16, and 32 threads. The report can include the names of team members on the cover page. In particular, focus on any performance differences that you notice between the chunking and striding methods of parallelization. Explain clearly in your report, the reason for these differences.

Note the following useful implementation tips for maximum speedup:

- *Parallelization process.* Parallelize the code that computes the updated value *new_x* during each iteration as well as the reduction process of calculating the SSD value.
- *Use of ping-pong buffers.* Note that the serial code uses two buffers — one to store *x* values from the previous Jacobi iteration and another to store *new_x* values calculated for the current iteration. In your parallel implementation, "ping-pong" between two buffers, one that stores values calculated during the previous iteration $k-1$ and another that is used to store values calculated during the current iteration k. You must never physically copy values between these buffers between iterations since that overhead will lead to tremendous slowdown. Rather maintain two pointers, say *prev* and *curr*, that point to these buffers; during iteration k, the *prev* pointer will point to the buffer containing values calculated during iteration $k-1$ (which you can treat as read only) and *curr* will refer to the operating buffer to which the updated values are being written to during iteration k. For the next iteration, swap the two pointers, and so on.
- *Use of barrier synchronization primitive.* Do not continually create and destroy threads each time through an iteration. That is, do not use *pthread_join()* as the mechanism for barrier synchronization between threads. Rather create the threads once and use the barrier synchronization implementation provided by the pthread library.