Iterative Jacobi Solver

Prof. Naga Kandasamy
ECE Department, Drexel University

This assignment, worth ten points, is due February 20, 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:

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

where the unknowns are $x_0, x_1, x_2, \ldots, x_{n-1}$. 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.

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.

Answer the following questions:

- (**5 points**) Complete the function *compute_using_avx()* to develop a SIMD version of the Jacobi solver using AVX instructions. You may develop additional code and data structures as needed.

- (**5 points**) Complete the function *compute_using_pthread_avx()* to develop a parallel version of the Jacobi solver using a combination of pthreads and AVX instructions. You may develop additional code and data structures as needed. Use pthreads to coarsely decompose the problem between multiple threads. Within each thread, use AVX instructions to extract fine-grained parallelism. For best performance, do not constantly create and destroy threads for each iteration; rather create them one and use the appropriate synchronization mechanisms to coordinate. Use ping-pong buffers to store values generated between two iterations.

The program given to you accepts the width of the square matrix and number of threads as command-line parameters. Solutions provided by the optimized implementations are compared to that generated by the reference code by printing out the relevant statistics.

Upload all source files needed to run your code on xunil as a single zip file on BBLearn. **Do not include any executable or object files in your submission.** 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 the pthreads version, report results for 4, 8, 16, and 32 threads. The report can include names of team members on the cover page.

Note. Comment out the *fprintf()* and *printf()* functions when measuring the execution times since these system calls incur high overhead.