Quang Anh Hoang (qh62@drexel.edu)

Iterative Jacobi Solver using OpenMP

ECEC 622: Parallel Computer Architecture
Professor Naga Kandasamy

1. Introduction

Similar to Gaussian Elimination, Iterative Jacobi solver is a matrix solving method for Ax = B equation, with a given n x n matrix A, a given n x 1 matrix B, and n x 1 matrix of variable x. However, its approach is different from obtaining row echelon form of A. Iterative Jacobi solver "iteratively" approximates and recalculates values of x's elements based on given matrices A and B until the change of x is smaller than a threshold value (10e-5 in this assignment). This is due to the matrix x will converge as the program finishes each iteration. Since the function Ax = B can be written in the following form:

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

then for an variable x_i of matrix x, its value can be computed using this equation deducted from the above system:

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

Therefore, the value of x can be found by iterating through calculating new value for each element in x matrix and find the difference, or MSE (Mean Squared Error) of the matrix until it is less than the threshold, and it is when the system converges to a small error and the result is close enough to the correct answer.

The main task of this assignment is parallelizing two tasks of the program: computing new values and calculating mean squared error, or "Sum of Squared Error" (SSD). Since SSD is the squared value of MSE, obtaining SSD would obviously result in obtaining MSE. At the beginning of each iteration, each thread computes the new value for an element in x matrix, then jump forward by a number, which is equal to number of threads, of elements (striding method) and computes the new value for the next element. One thread finishes computing new values when all assigned elements have new value, then it moves to the next task: accumulate the SSD. It goes through every element in x that it just calculated, gets the difference between the old value and the new value, then squares it and accumulates to a local variable named "partial difference", which eventually will be accumulated to SSD. A mutex lock is used to make sure only one thread would access the SSD value at a moment of time.

2. Implementation Design

2.1. Jacobi function:

Although Jacobi solver algorithm is similar to previous assignment with pthread library, using OpenMP actually simplifies the implementation a lot. As described above, the Iterative Jacobi Solver function approximates the x matrix and calculate the mean squared error value; if this value is smaller than a threshold (10e-5), the function repeats its approximation and calculating MSE. Similar to the pthread Jacobi Solver program, this program with OpenMP also uses two pointers (float *src and float *dest) to alternate between two arrays of x, which helps reduce runtime when switching an array from the source to the destination array.

```
if ((num_iter % 2) == 0) {
        src = mt_sol_x.elements;
        dest = new_x.elements;
}
else {

        dest = mt_sol_x.elements;
        src = new_x.elements;
}
```

The for loop, in which the new values in x array is calculated and stored to, is parallelized using OpenMP as following:

```
#pragma omp parallel num_threads(num_threads) private(i, j, sum)
{
        #pragma omp for
        for (i = 0; i < num_row; i++) {
              sum = 0.0;
              for (j = 0; j < num_col; j++) {
                    if (i != j) {
                          sum += A.elements[i*num_col + j]*src[j];
                    }
              }
              dest[i] = (B.elements[i] – sum)/A.elements[i*num_col+i];
        }
}
```

As indicated in the above code, within each iteration, all elements in src array, except for one at row (i), are multiplied with coefficients on row (i) in matrix A then accumulates to get a value named “sum”. After that, the next value of x(i) is calculated using the equation (1) given above. The SSD value is also calculated by iterating through each elements in both source and destination arrays, find the differences, square them and accumulate. This value is then taken the square root to obtain MSE value, and once MSE is smaller than threshold, the done value is set to 1 and the program finishes.

3. Result

There are twelve different configurations used in this assignment to examine the performance of parallelizing Gaussian Elimination, with four different values of number of threads: 4, 8, 16 and 32; three different values of matrix size: 512 x 512, 1024 x 1024 and 2048 x 2048. Each configuration is run for 20 times and speedups are recorded and shown in the tables below for each method:

	4 threads	8 threads	16 threads	32 threads
512 x 512	3.429357	2.714223	2.243530	1.387390
1024 x 1024	6.207686	4.449688	3.452370	3.032511
2048 x 2048	6.848023	5.849196	5.934124	6.171044

Table 1: Average speedup of multi-thread Iterative Jacobi function, striding method

According to the collected result, the speedup of multi-thread program is enormously large and it shows a very good improvement of performance, since all of configurations results in speedup larger than 1. As the matrix’s size increases, the speedup of program doubles (from 3.43 times at 512 x 512 to 6.85 times at 2048 x 2048). And for every matrix dimension, 4-thread program shows the most significant speedup as it is the most effective with lowest cost of creating threads while keeping runtime low. On average, using 4 threads could accelarate the program by 243% with 512 x 512 matrix; for 1024 x 1024 matrix, it could make the program runs at 6.21 times faster, and for 2048 x 2048, this number is 6.85 times. This shows that multithreading obviously and massively improve the program’s performance and as the volumn of data (which is proportional with the size of matrix A) increases, the speedup’s coefficient also increases and the program takes much less time to complete.