ECE 420 Parallel and Distributed Programming Lab 3: Solving a Linear System of Equations via Gauss-Jordan Elimination using OpenMP[*]

Winter 2018

1 Background

Consider the problem of solving a linear system of equations

$$\begin{aligned}
a_{11}x_{11} + a_{12}x_{12} + ... + a_{1n}x_{1n} &= b_1 \\
a_{21}x_{21} + a_{22}x_{22} + ... + a_{2n}x_{2n} &= b_2 \\
... &\quad ... \\
a_{n1}x_{n1} + a_{n2}x_{n2} + ... + a_{nn}x_{nn} &= b_n.
\end{aligned}$$

Denote the coefficient matrix by

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix},$$

the constant vector by

$$\vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix},$$

[*]In this manual, all the indeces start from 1.

and the unknown variables by

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix}.$$

The linear system of equations can be represented by

$$\mathbf{A} \cdot \vec{x} = \vec{b}.$$

The linear system of equations is actually characterized by the augmented matrix $\mathbf{G}$,

$$\begin{aligned} \mathbf{G} &= \{\mathbf{A}|\vec{b}\} \\ &= \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right). \end{aligned}$$

Any operation on the original system of equations is equivalent to performing some corresponding operation on the augmented matrix $\mathbf{G}$. And an augmented matrix can easily be mapped back to a linear system of equations. For simplicity, we will perform calculation on the augmented matrix.

There are 3 types of linear operations (or row operations) which will not change the solution(s) to the linear system of equations:

1. interchanging any two rows;

2. multiplying each element of a row by a nonzero constant;

3. replacing a row by the sum of itself and a constant multiple of another row in the augmented matrix.

For these row operations, we use the following notations:

1. $R_i \leftrightarrow R_j$: interchange the i^{th} row and the j^{th} row;

2. αR_i: multiply each element of row i by a nonzero α;

3. $R_i + \alpha R_j$: replace row i with the sum of row i and α times row j.

To solve the linear system of equations, the basic idea is to transform the original linear system to an equivalent new system via some linear operations. And the new system should be reduced to some good form such that every equation in the system has exactly one different variable with a nonzero coefficient, from which we can simply "read" the solutions. In other words, the target augmented matrix should be in the following form:

$$\left(\begin{array}{cccc|c} d_{11} & 0 & ... & 0 & b'_1 \\ 0 & d_{22} & ... & 0 & b'_2 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & d_{nn} & b'_n \end{array}\right).$$

The Gauss-Jordan Elimination is a procedure to achieve the above goal.

1.1 Gaussian Elimination with Partial Pivoting

Gaussian Elimination will transform the augmented matrix into its equivalent "upper triangular" form, in which the elements below the main diagonal are all zeros. It iteratively eliminates the elements below the main diagonal from the first column to the last column via row operations. Algorithm 1 describes such a procedure.

Note that the partial pivoting part is important in this procedure. It will prevent the case in which u_{kk} is zero or close to zero. Thus, with partial pivoting, the program will be more numerically stable. Also, note that in the row replacement operation, j starts from k, since the first $k-1$ elements are always zero in this algorithm.

1.2 Jordan Elimination

After we obtain an "upper triangular" $\mathbf{U}$ from Gaussian Elimination, the Jordan Elimination can further transform it into the desired form. Similarly, the basic idea is to iteratively eliminate the elements *above* the main diagonal for each column, one after another.

The inner for loop performs the row replacement. However, for each row, we only need to update the two elements d_{ik} and $d_{i(n+1)}$, since elements on the other columns actually stay the same. See the example for an illustration.

After we get $\mathbf{D}$, we can compute the desired solution $\vec{x}$ simply by

$$x_i = d_{i(n+1)}/d_{ii}, \text{for any } i.$$

Algorithm 1 Gaussian Elimination

Input: An augmented matrix $\mathbf{G} = \{\mathbf{A}|\vec{b}\}$, where $\mathbf{A} = (a_{ij})$ is an $n \times n$ matrix and $\vec{b} = (b_i)$ is an n-dimensional vector.

Output: The augmented matrix $\mathbf{U}$ (the elements are denoted as u_{ij}) that is equivalent to $\mathbf{G}$ and is in the "upper triangular" form.

```
Initially, $\mathbf{U} \leftarrow \mathbf{G}$
for $k = 1$ to $n - 1$ do/*eliminate elements below the diagonal to zero one column after another*/
    /*Pivoting*/
    In $\mathbf{U}$, from row $k$ to row $n$, find the row $k_p$ that has the maximum absolute value of the element in the $k^{th}$ column
    Swap row $k$ and row $k_p$
    /*Elimination*/
    for $i = k + 1$ to $n$ do
        $temp = u_{ik}/u_{kk}$
        for $j = k$ to $n + 1$ do
            $u_{ij} \leftarrow u_{ij} - temp \cdot u_{kj}$/*row replacement*/
        endfor
    endfor
endfor
```

Algorithm 2 Jordan Elimination

```
Input: The output of the Gaussian Elimination U (an $n \times (n+1)$ matrix).
Output: The augmented matrix D (with elements denoted by $d_{ij}$) that is equivalent to G and is in our final target form.

Initially, $\mathbf{D} \leftarrow \mathbf{U}$
for $k = n$ to 2 do/*eliminate elements to zero for each column one after another*/
    for $i = 1$ to $k - 1$ do/*row replacement one row after another*/
        $d_{i(n+1)} \leftarrow d_{i(n+1)} - d_{ik}/d_{kk} \cdot d_{k(n+1)}$
        $d_{ik} \leftarrow 0$
    endfor
endfor
```

1.3 An Example

We will give an example to demonstrate the described algorithms. Consider a linear system of equations:

$$\begin{aligned} 2x_{11} + 4x_{12} - 2x_{13} &= 3 \\ -4x_{21} - 8x_{22} + 5x_{23} &= -4 \\ 4x_{31} + 4x_{32} - 5x_{33} &= 4. \end{aligned}$$

The corresponding augmented matrix is

$$\left(\begin{array}{ccc|c} 2 & 4 & -2 & 3 \\ -4 & -8 & 5 & -4 \\ 4 & 4 & -5 & 4 \end{array}\right)$$

The Gauss-Jordan Elimination with partial pivoting on it will be

$$
\left(\begin{array}{ccc|c} 2 & 4 & -2 & 3 \\ -4 & -8 & 5 & -4 \\ 4 & 4 & -5 & 4 \end{array}\right)
$$

$$
\xrightarrow{R_1 \leftrightarrow R_2} \left(\begin{array}{ccc|c} -4 & -8 & 5 & -4 \\ 2 & 4 & -2 & 3 \\ 4 & 4 & -5 & 4 \end{array}\right) \text{(pivoting)}
$$

$$
\xrightarrow{R_2 + \frac{1}{2}R_1} \left(\begin{array}{ccc|c} -4 & -8 & 5 & -4 \\ 0 & 0 & \frac{1}{2} & 1 \\ 4 & 4 & -5 & 4 \end{array}\right)
$$

$$
\xrightarrow{R_3 + R_1} \left(\begin{array}{ccc|c} -4 & -8 & 5 & -4 \\ 0 & 0 & \frac{1}{2} & 1 \\ 0 & -4 & 0 & 0 \end{array}\right)
$$

$$
\xrightarrow{R_2 \leftrightarrow R_3} \left(\begin{array}{ccc|c} -4 & -8 & 5 & -4 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 \end{array}\right) \text{(pivoting; it happens to be the end of Gaussian Elimination)}
$$

$$
\xrightarrow{R_1 - 10R_3} \left(\begin{array}{ccc|c} -4 & -8 & 0 & -14 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 \end{array}\right) \text{(Starting Jordan Elimination)}
$$

$$
\xrightarrow{R_1 - 2R_2} \left(\begin{array}{ccc|c} -4 & 0 & 0 & -14 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 \end{array}\right)
$$

2 Task and Requirement

Task: Using OpenMP, implement a program to solve linear systems of equations by Gauss-Jordan Elimination with partial pivoting. The input will be a coefficient matrix $\mathbf{A}$ and a vector $\vec{b}$. The output will be a vector $\vec{x}$, where

$$\mathbf{A} \cdot \vec{x} = \vec{b}.$$

Requirements and Remarks:

- Use the scripts in “Development Kit Lab 3” to generate input data, load data and save results. Refer to the *ReadMe* file for details on how to use them.
- Time measurement should be implemented.
- The number of threads should be passed as the only command line argument to your program.
- *Optimize* the performance of your implementation as much as possible using the techniques learned in class.
- You don’t need to consider the *singular* cases, i.e., a linear system with no solution or an infinite number of solutions. The input data generated by “datagen.c” will avoid such cases. You *do* need to include the partial pivoting procedure in your code to make the computed results correct and numerically stable.

Lab Report Requirements:

1. Describe your implementation clearly.
2. Compare and discuss the performance (speedup or efficiency) of your implementation under different numbers of threads used. Explain your results.
3. Present the design choices and performance optimization analysis in your lab report. In the report, show the run times of your optimized code compared against some baseline inferior version(s). Use figures and/or tables to show

the results under various setups (e.g., different scheduling policies and chunk sizes, different parallelization strategies, and different numbers of threads and so on) and under sufficiently large input(s). In other words, an effort for performance optimization must be demonstrated in your lab report.

4. Please also refer to the "Lab Report Guide" for other requirements.

Submission Instructions:

Each team is required to submit BOTH a hard copy of printed lab report to the assignment box AND the source code on eClass. The report should be submitted in the assignment box on the 2nd floor - pedway between ICE and ETLC. Please check eClass for the code and report submission due date.

For code submission, each team is required to submit a zip file to eClass. The zip file should be named "StudentID-Hxx.zip", where "StudentID" is the Student ID of **one** of your group members (doesn't matter which member) and "Hxx" is the section (H41, H11 or H42).

The zip file should contain the following files:

1. "readme": a text file containing instructions on how to compile your source files;
2. "members": a text file listing the student IDs of ALL group members, with each student ID occupying one line;
3. "Makefile": the makefile to generate the executable. Please ensure that your Makefile is located in the root folder of that zip file and the default "$make" command will generate the final executable program named "main";
4. All the necessary source files to build the executable "main";

DO NOT include the compiled data generation file, "serialtester" file, or the input/output data file.

Note: you MUST use the file names suggested above. File names are case-sensitive. You MUST generate the required zip file by directly

compressing all the above files, rather than compressing a folder containing those files.

3 Hints and Tips

- To improve the performance, you might need to find out all components of the program that each can be run in parallel first. Try to reduce the number of forks and implicit joins due to repeated uses of parallel directives. In other words, try to reuse the team of threads launched by a parallel directive.
- What kind of scheduling policies will yield the best result for each OpenMP for loop?
- How many threads should be used?
- Should we use the same number of threads to handle each for loop? Or should we vary the number of threads dynamically at different stages of the program? Should we vary the scheduling policy, chunksizes and other parameters?
- To guarantee the correctness, how should we effectively protect a critical section if there is any?
- Would the single or master directive be useful here?
- Is it helpful to use the collapse clause if there is any nested loop?
- In the pivoting procedure, swap the index or pointers rather than swapping all the row elements in memory.
- The *debugging* is similar to Pthread debugging. When compiling, instead of using the "-g" flag, you might need to use the "-ggdb" flag to make the debugger work well on an OpenMP multi-threaded program. You can try other flags like "-gstabs", "-gstabs+". However, whether they would work depends on the system.

Appendix

A Marking Guideline (to be posted)