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

Winter 2022

1 Background

Consider the problem of solving a linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\ ... &\quad ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &= b_n. \end{aligned}$$

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 indices start from 1.

and the unknown variables by

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

The linear system of equations can be represented by

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

The linear system of equations can also be characterized by the augmented matrix **G**,

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

Any operation on the original system of equations is equivalent to performing some corresponding operation on the augmented matrix **G**. And an augmented matrix can easily be mapped back to a linear system of equations. There are 3 types of linear operations (or row operations) which will not change the solution(s) of the linear system of equations:

1. interchanging any two rows;
2. multiplying each element of a row by a nonzero scalar;
3. adding onto a row with a scalar multiple of another row.

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. $R_i + \alpha R_j$: add onto row i with the product between scalar α and row j.

To solve the linear system of equations, the basic idea is to transform the original linear system into an equivalent new system via some linear operations. The new system should be reduced to a good form such that every equation (row) in the system has exactly one variable with 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 this 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 this procedure.

Note that the partial pivoting part is important in this procedure. It will prevent the case where 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, Jordan Elimination can further transform the matrix into its final diagonal form. The basic idea is to iteratively eliminate the elements *above* the main diagonal for each column, one after another. Algorithm 2 describes this procedure.

Note that the inner for loop performs 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 $\mathbf{U}$ (an $n \times (n+1)$ matrix).
Output: The augmented matrix $\mathbf{D}$ (with elements denoted by $d_{ij}$) that is
equivalent to $\mathbf{G}$ 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 give an example to demonstrate the described algorithms. Consider a linear system of equations:

$$\begin{aligned} 2x_1 + 4x_2 - 2x_3 &= 3 \\ -4x_1 - 8x_2 + 5x_3 &= -4 \\ 4x_1 + 4x_2 - 5x_3 &= 4. \end{aligned}$$

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. Specifically, compile and run "datagen.c" to generate the input data. For marking purpose, use the function "Lab3LoadInput" and "Lab3SaveOutput" to load your input data and save your output data.
- Time measurement should be implemented. Note that it should not include the matrix loading or saving time.
- 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 do not 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.
- Test your final solution on the VM cluster and make sure it works there.

Hint: You can verify your saved result against the result from a serial version of the program by compiling and running "serialtester.c".

Lab Report Requirements:

1. Describe your best implementation clearly.
2. Explain any other optimization methods (whether successful or not) you have attempted. This is to demonstrate that you have made an effort in optimizing your code.
3. Discuss the performance (speedup or latency) of your best implementation with different numbers of threads used. Explain your results.
4. Compare the execution time of your optimized code against some inferior baselines (alternative optimization strategies). Use figures and/or tables to show the performance under various setups (e.g., different scheduling policies and chunk sizes, different parallelization strategies, different numbers of threads, etc.).
5. Please also refer to the "Lab Report Guide" for other requirements.

Submission Instructions:
Each team is required to submit a zip file to eClass by the submission deadline. The zip file should be named "StudentID.zip", where "StudentID" is the Student ID of **one** of your group members (it does not matter which member, though this should be consistent for all submissions throughout the course). The zip file should contain the following three folders:

1. "Code": this folder should contain all the necessary codes to compile the solution executable, including:
   (a) "Makefile": the makefile to generate the executable. By executing the "make" command, the executable "main" should be generated. This should be your **most optimized implementation**.
   (b) All the necessary source files to build your submitted implementation. You do not need to submit any of the baseline or alternative implementations that are presented in your lab report.
2. "Members": this folder should contain a single text file *named* "members.txt", listing the student IDs of ALL group members, with each student ID occupying one line.

3. “Report”: this folder should contain the lab report, saved as a single PDF file *named* ”lab_report.pdf”.

DO NOT include any input/output data file or compiled executable.

Note: you must use the file names suggested above. File names are case-sensitive. Please generate the required zip file by directly compressing all of the aforementioned folders, rather than compressing a parent folder containing these folders as subfolders.

3 Hints and Tips

- To improve the performance, you should find out all of the components in the program that 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 the 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 correctness, how should we protect the critical sections (if there are any)?
- Would the single or master directive be useful here?
- Is it helpful to use the collapse clause if there are nested loops?
- In the pivoting procedure, it may be better to swap the indices or pointers rather than swapping all the row elements in the 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 depend on the system.

General Marking Guideline

Code (4 Marks):

- Correct implementation
- Speedup from optimization

Report (6 Marks):

- Clear description of the best implementation
- Demonstrate efforts in optimization
- Improvements over the baseline versions
- Discussion and explanation